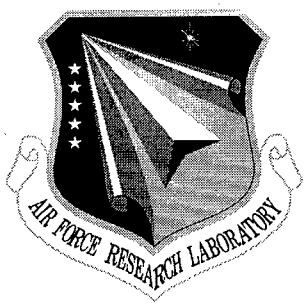
RL-TR-97-220
Final Technical Report
February 1998



FUNCTIONAL REPRESENTATION OF SOFTWARE SYSTEMS AND COMPONENT-BASED SOFTWARE TECHNOLOGY

Ohio State University Research Foundation

Sponsored by
Advanced Research Projects Agency
ARPA Order No. A714

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

AIR FORCE RESEARCH LABORATORY
ROME RESEARCH SITE
ROME, NEW YORK

This report has been reviewed by the Air Force Research Laboratory, Information Directorate, Public Affairs Office (IFOIPA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

RL-TR-97-220 has been reviewed and is approved for publication.

APPROVED:

DOUGLAS A. WHITE
Project Engineer

FOR THE DIRECTOR:

WARREN H. DEBANY, Technical Advisor
Command, Control & Communications Directorate

If your address has changed or if you wish to be removed from the Air Force Research Laboratory mailing list, or if the addressee is no longer employed by your organization, please notify AFRL/IFTD, 525 Brooks Rd, Rome, NY 13441-4505. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document require that it be returned.

ALTHOUGH THIS REPORT IS BEING PUBLISHED BY AFRL, THE RESEARCH WAS ACCOMPLISHED BY THE FORMER ROME LABORATORY AND, AS SUCH, APPROVAL SIGNATURES/TITLES REFLECT APPROPRIATE AUTHORITY FOR PUBLICATION AT THAT TIME.

FUNCTIONAL REPRESENTATION OF SOFTWARE SYSTEMS AND COMPONENT-BASED SOFTWARE TECHNOLOGY

B. Chandrasekaran
Bruce Weide

Contractor: Ohio State University
Contract Number: F30602-93-C-0243
Effective Date of Contract: 1 October 1993
Contract Expiration Date: 31 March 1997
Program Code Number: 6D10
Short Title of Work: Functional Representation of Software Systems
Period of Work Covered: Oct 93 – Mar 97

Principal Investigator: B. Chandrasekaran
Phone: (614) 292-0923
RL Project Engineer: Douglas A. White
Phone: (315) 330-2129

This research was supported by the Advanced Research Projects Agency of the Department of Defense and was monitored by Douglas A. White, AFRL/IFTD, 525 Brooks Rd, Rome, NY.

REPORT DOCUMENTATION PAGE	Form Approved OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	February 1998	Final Oct 93 - Mar 97

4. TITLE AND SUBTITLE

FUNCTIONAL REPRESENTATION OF SOFTWARE SYSTEMS AND COMPONENT-BASED SOFTWARE TECHNOLOGY

5. FUNDING NUMBERS

C - F30602-93-C-0243
PE - 61101E
PR - A714
TA - 00
WU - 01

6. AUTHOR(S)

B. Chandrasekaran and Bruce Weide

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

The Ohio State University Research Foundation
1960 Kenny Road
Columbus, OH 43210

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Advanced Research Projects Agency
3701 North Fairfax Drive
Arlington, VA 22203-1714

AFRL/IFTD
525 Brooks Rd.
Rome, NY 13441-4505

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

RL-TR-97-220

11. SUPPLEMENTARY NOTES

AFRL Project Engineer: Douglas A. White/IFTD/(315) 330-2129

12a. DISTRIBUTION AVAILABILITY STATEMENT

Approved for public release; distribution unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

The overall objectives of this project were to develop approaches to program comprehension that would provide significant added benefits in many aspects of software engineering. As one part of that effort, the RESOLVE/ACTI framework for a software component composition technology was developed. The technology focuses on development of software components that can be reused and a composition discipline that helps in creating programs whose properties can be efficiently reasoned about. As another part of the effort, a device comprehension framework called Functional Representation was applied, and its utility shown for software architecture comprehension, legacy software reengineering and representation of system requirements.

14. SUBJECT TERMS

software, software understanding, software development environments, reuse, reengineering

15. NUMBER OF PAGES

218

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

Standard Form 298 (Rev. 2-89) (EG)
Prescribed by ANSI Std. 239.18
Designed using Perform Pro, WHS/DIOR, Oct 94

Table of Contents

1. Introduction

This report outlines the progress made in the project entitled "Functional Representation of Software Systems and Component-Based Software Technology," during the project period, October 1, 1993 to March 31, 1997. In order to give context, we restate the goals, describe the underlying framework, and give a summary of progress. Many of the detailed results are described in technical reports and papers, some of which were published during the last three years, while others are in the process of being submitted or considered for publication. The most important of these are attached as appendices to the report.

1.1. Goals of the Project

Our project was organized around two related top-level goals:

- To develop a discipline and technology of composing reliable software from parts
- To develop a framework for program comprehension, and to apply it for software maintenance, reuse and reengineering.

Comprehension and design are dual enterprises. We understand an artifact by recognizing functional modules or components, and building a picture of how the components work together to create the functions of the artifact. If an artifact (or natural system) is so constructed that the component interactions cannot be easily reasoned about, our comprehension of the artifact is correspondingly limited, resulting in either incomplete or incorrect predictions of the artifact's behavior. Conversely, we design artifacts by composing parts, whose behavior we understand, into larger systems. We reason about the behavior of the composed artifact to assure ourselves that the artifact realizes intended behavior. How easily we can understand a design depends on two things: how well we understand what the components do, and whether the composition was done in such a way that component interactions can be easily reasoned about.

The project was largely organized as two interacting subprojects.

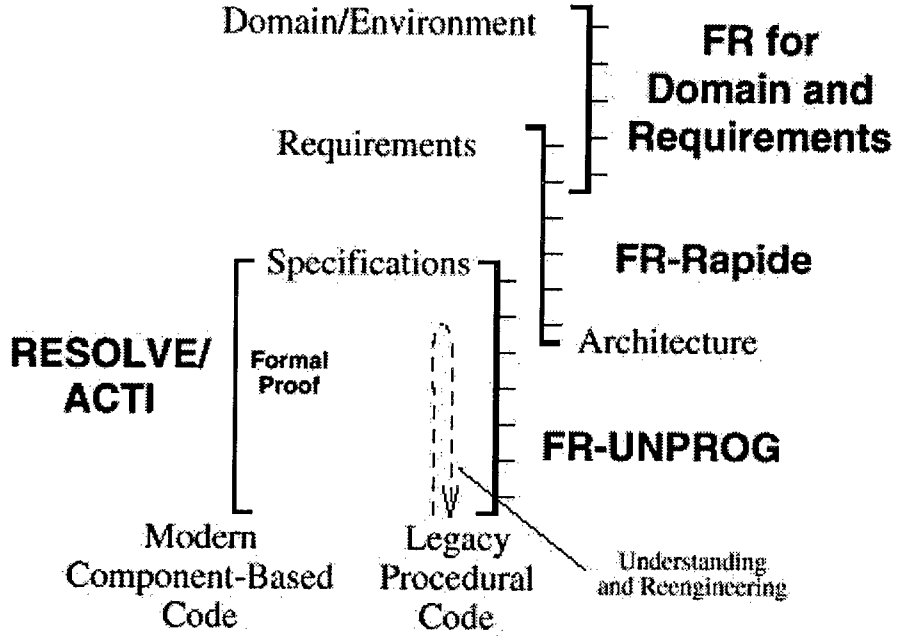
1. The **RESOLVE/ACTI project** focused on developing both the theoretical foundations for composable software as well as implementations of component libraries with the right properties. The ACTI theory develops a software understanding framework that is especially useful in the software composition task.
2. The **Functional Representation project** focused on the use of FR – a specific family of representations for software comprehension – for a number of tasks in software engineering other than composition. Specifically, we explored the application of FR to requirements engineering, explanation of software architectures, and reengineering of legacy software.

As one would expect, the software understanding parts of the two projects interacted closely. The ACTI and FR frameworks share common points of view, but because of the difference in applications, focus on different aspects of understanding. FR is a general theory of comprehension of causal systems, while ACTI's focus is on the mathematical framework in which to understand software components.

Our work on the project spans a broad range of software engineering goals, processes, and artifacts in these contexts:

1.) Component-Based Software
2.) Systems, Domain, and Requirements Engineering
3.) Software Architecture
4.) Legacy Software.

In each of these contexts, we developed and characterized technologies to capture and exploit human understanding of software artifacts. This figure shows these contexts and locates our work.



The vertical dimension displays semantic levels and software engineering artifacts. At the top, most removed from code, are domain and environment objects and concepts. Going downward, there are commitments, artifacts, and concepts that are increasingly code-related - requirements, specifications, architecture, and code.

Our work in FR for domain and requirements captures and formalizes understanding of the environment, domain objects, and system requirements. This understanding is hierarchical, containing refinements and languages at various abstraction levels.

FR-Rapide captures intentions and their implementation in an architecture, specifically an architecture called Rapide. It is similarly hierarchical, and intentions represented involve requirements, specifications, and architecture design down to architecture description language code.

RESOLVE/ACTI contains both specifications and their implementation in modern component-based code. Its captured understanding includes complete abstract behavior, given by formal specifications, and justification that the behavior is realized in code, shown by proof.

FR-UNPROG represents understanding of how specifications are implemented in real-world legacy software. It focuses on hierarchical understanding used in human and automatic program understanding and reengineering.

1.2. Organization of the Report

Much of our work has been reported in the literature in the form of papers in journals and conferences, and technical reports that are in the process of being converted to publications. We devote the next section to a brief account of some of the central theoretical ideas. We follow this with a section that outlines progress that we have made in specific areas and their significance. Finally, we attach as appendices a selection of the most important publications and technical reports, where the reader can find the technical details of the work.

2. Overview of the Theoretical Frameworks

2.1. RESOLVE/ACTI Framework for Component-Based Software

2.1.1. RESOLVE

RESOLVE refers to three related things: a *conceptual framework* to guide thinking about component-based software systems; a *specific language* to support easy description of components and systems within that framework; and a *general discipline* for using that language (or others with comparable features) to design high-quality software components and systems. RESOLVE software components are parameterized modules. A typical specification module formally defines the structural interface and functional behavior of an encapsulated abstract data type (ADT), and associated operations whose parameters are of that type and other types. A typical implementation module describes how such a type is represented as a composition of other ADTs, and how the associated operations are effected by invoking sequences of operations associated with the types used in the representation.

At first glance, the basic RESOLVE framework and notation resemble those of modern formal specification languages (e.g., Z, Larch) and object-based or object-oriented programming languages (e.g., Ada, C++, Eiffel, ML). What RESOLVE provides beyond its integrated language is an entire framework in which all the important, but sometimes conflicting, aspects of software design can be considered at once. In this framework, component engineering can exploit new perspectives that differ from conventional approaches in many important ways. The full compass of advantages can best be appreciated by examining specific reusable software components. This is why we have spent considerable effort designing, implementing, and using a variety of reusable components. They are specified and implemented in RESOLVE, and — to facilitate technology transfer — also implemented in Ada and/or C++.

2.1.2. ACTI Model of Software Understanding

The ACTI model is centered on the notion of a "software subsystem," a generalization of the idea of a module or a class that serves as the building block from which software is constructed. A subsystem can vary in grain size from a single module up to a large-scale generic architecture. ACTI is designed specifically to capture the larger meaning of a software subsystem in a way that contributes to human understanding, not just the information necessary to create a computer-based implementation of its behavior. The ACTI model is based on four different kinds of subsystems:

- *Abstract Instance* – A disembodied subsystem specification or interface description. There is no implementation associated with anything defined in the specification.
- *Concrete Instance* – A subsystem that provides implementations for its types and operations. All of the defined types and operations in the subsystem are represented and/or implemented.

- *Abstract Template* – A subsystem-to-subsystem function that, when applied to its argument, which is some abstract instance, will generate another abstract instance. Effectively, an abstract template is a form of generic subsystem specification.
- *Concrete Template* – A subsystem-to-subsystem function that, when applied to its argument, which is some concrete instance, will generate another concrete instance. Thus, a concrete template is a form of generic subsystem implementation.

The name "ACTI" is an acronym derived from these four terms: "Abstract and Concrete Templates and Instances." The distinction between "abstract" and "concrete" embodies the separation between a specification or interface, and an implementation or representation. The distinction between "template" and "instance" allows one to talk about both generic subsystems and the product of fixing (binding) the parameters of such a generic subsystem: an instance subsystem. Formally, ACTI is a collection of mathematical spaces, together with relations and functions on those spaces, that can be used in explaining (or defining) the denotational semantics of program constructs. In spirit, the model was developed in accordance with the denotational philosophy. In the denotational philosophy the program, or program fragment, is first given a semantics as an element of some abstract mathematical object, generally a partially ordered set. The semantics of the program are a function of the semantics of its constituent parts; properties of the program are then deduced from a study of the mathematical object in which the semantics lives. ACTI is not a programming language, however. Instead, it is a mathematical model that is useful for programming language designers, or researchers studying the semantics of programming languages. It is a formal, theoretical model of the structure and meaning of software subsystems. It is rich enough to be used when designing new languages, and has been shown to subsume the run-time semantic spaces of several existing languages chosen to be representative of the modern imperative, OO, and functional philosophies. ACTI has two features that specifically address the inadequacies described in the introduction:

1. In ACTI, a software subsystem (building-block) has an intrinsic meaning; it is not just a syntactic construct used for grouping declarations and controlling visibility. This meaning encompasses an abstract behavioral description of all the visible entities within a subsystem.
2. The meaning of a software subsystem is not, in general, hierarchically constructed. In fact, it is completely independent of all the alternative implementations of the subsystem.

Thus, ACTI provides a mechanism for describing what a subsystem does, not just how it is implemented. The meaning provided for a subsystem is a true abstraction – a "cover story" that describes behavior at a level appropriate for human understanding without explaining how the subsystem is implemented. Further, ACTI provides a formally defined mechanism, called an interpretation mapping, that captures the explanation of why an implementation of a subsystem will give rise to the more abstractly described behavior that comprises the meaning attributed to the subsystem – in short, an explanation for why the cover story works.

2.2. Functional Representation Framework for Artifact Understanding

For a number of years, one of the PI's (Chandrasekaran) has been engaged in research whose goal is to understand understanding, particularly understanding of how an artifact "works." This research has produced a framework called Functional Representation, which is one of the technical bases for the project. In this framework, comprehending the functioning of an artifact consists of producing the following descriptions for it:

- the *function(s)* of the artifact
- the *structure* of the artifact, i.e., a specification of its components and how they are put together
- an account of how the artifact achieves its function based on the roles played by the components and general laws that pertain to the domain.

Functional Representation (FR) is a general language and framework for representing functions, structure and the causal processes that underlie the operation of the device. The FR framework has been applied mostly to engineered physical devices. However, in our earlier work we have shown that FR is applicable to understanding abstract devices such a programs and logistic plans. In software engineering, the language of plans has often been proposed as the basis for encoding comprehension. FR subsumes programming plans, in that it contains the same information as plans do in general, but also adds further information to make them fully satisfy the desiderata for comprehension. In particular, plans as currently represented in software engineering satisfy to some degree parts 1 and 2 of the desiderata. FR satisfies part 3 as well.

There are different ways of providing an account of how the function is achieved, the third item in the above list. Mathematically, it is required is to show that the function of the artifact can be derived from the component properties and the way the components are composed. For certain purposes, an account giving relevant state changes of the artifact is useful. In this account, a series of partial state descriptions of the artifact is given. The initial state corresponds to the starting conditions, and the final state corresponds to the predicate that describes the function. Each intermediate state change is explained by appealing to the function of some component. It is explained by showing which function of which component played the causal role in the transition, and by showing the precise way in which the state change happened. This kind of explanation has been used traditionally in the FR work. This form of accounting is useful for explanation to human beings, and for certain kinds of problem solving activities, such as debugging and design criticism. Logically, other kinds of demonstrations can be substituted. For example, the RESOLVE/ACTI framework seeks to ensure that the composed program satisfies the requirements by developing an effective modular proof technique.

The RESOLVE/ACTI framework shares a number of intuitions about comprehension with FR, although it has used somewhat different terminology. Because of its exclusive focus on software, RESOLVE has a much better developed formalism for representing the relevant aspects of programs, and techniques for reasoning about composition. Because FR is a general framework for causal understanding, it needs domain-specific

theories when it is applied to particular domains. RESOLVE provides such a theory for the software domain.

3. Outline of Results

In this section, we summarize the results of our research. This section is organized around the contexts that we mentioned in the Introduction.

3.1. Component-Based Software

Our first context is component-based and high-quality software. Reusable components consist of complete formal specifications and implementing code. They are engineered for correctness, flexibility, efficiency, understandability etc. Work in component-based software applies generally to new software development with careful design and formal specifications.

RESOLVE/ACTI represents understanding of code consisting of formal specifications and proofs that the specifications are implemented in the code. There is a long history of research into representing such understanding. Mathematical approaches for representing specifications and proofs are relatively well-established. However, these approaches have not been applied in practical software engineering environments, especially for reusable components. Important issues include the design of productive programming systems that support correct specification, and the explanatory quality of specifications and proofs.

RESOLVE/ACTI is an especially complete and robust framework for component-based software. It includes goals, language, discipline, a semantic model, and a component library for component-based software. Our work has developed and characterized this approach with respect to component design and implementation, and with respect to model-based specification and proof quality.

Work with *iterator* abstractions illustrates RESOLVE/ACTI components and characterization. The simplest kind of iterator permits a client program to examine each item of a collection in some order, e.g. to accumulate information about set items, or print items in a tree. We give an interface model of iterators for arbitrary generic collections. The model is characterized with respect to both correctness provability and efficiency. For example, RESOLVE/ACTI's use of the swapping paradigm is shown to permit modular proof of correctness (unlike pointer copying), while preserving efficiency (unlike structure copying).

Specific contributions and accomplishments in this software engineering context include:

- Empirical studies showing benefits from RESOLVE black-box reuse
- The RESOLVE/C++ programming discipline, which allows one to build modularly certifiable/verifiable software components in C++
- A safe method for using white-box code inheritance
- The ACTI model of component-based systems and their semantics
- Specification and proof characterization, including use of observability and controllability, and abstraction relations.

Appendices 1 to 9 describe in detail the specific contributions summarized above.

3.2. FR Applications for System Engineering, Domain Modeling and Requirements Capture

Many military and commercial applications —e.g., weapon systems, air traffic control systems, or even a single aircraft or tank — are *heterogeneous* systems. They are assemblages of subsystems from many different domains, e.g., mechanical, electrical, thermal, and software systems. When high-level design is performed using abstract functions, perhaps represented by blocks in a diagram, the blocks may represent physical or software systems. FR is an appropriate framework for representing and reasoning about such systems. Building such systems also requires a language in which to acquire system requirements.

3.2.1. Causal and Functional Models of Objects and Reasoning about Their Composition

We made significant progress in this problem area. We looked at the following issues.

- Representing causal models of objects so that the behaviors of configurations of interacting objects can be reasoned about.
- Representing functions of device configurations.
- Representing functional requirements for design.

The technical issues addressed include:

- *Representation of an object*: its properties and property relations, ports at which an object can be connected to other objects, and how ports are loci of causal interactions. Objects are represented in points of view, which select certain properties for representation. Views also specify the object in the context of some generic environments, i.e., an object representation is with respect to the kinds of objects it can be in a causal relationship with. Properties can be static or dynamic. When they are dynamic, they are called state variables. Behavior of an object is the trajectory of the values of its selected state variables over time. Property relations are called behavioral specifications when the properties involved are the state variables. This basic ontology -- objects, static and dynamic properties, ports and behavioral specifications -- provides the basic primitives needed to represent individual objects in specific environments and reason about their properties and behaviors under various conditions.
- *Composing objects and deriving the properties and behavior of composed objects.* Ports are the basic "connection" points. Thus the structure of composed objects is given by specifying objects and the ports which form the connections between them. A basic framework for composing the behavioral specifications of the component objects into a set of behavioral specifications for the composite object has been developed.
- *Various means of abstracting behaviors* of the composite objects so that behavioral abstractions at new levels of description can be introduced. This is important because one of the most important consequences in assembling components is that some new behaviors can be described effectively only by using new primitive terms. For

example, at some point a composition of transistors and resistors becomes an "adder," whose behavior is described not in terms of currents and voltages, but in terms of addends and sums. Representing the properties and property relations of the composed object may get arbitrarily complicated, especially if we wish to focus on new properties that are not part of the descriptions of the individual objects.

- *Structural explanation.* Given a composite object and a set of behaviors that it exhibits, there are different ways of "explaining" the behaviors. One kind of explanation is to appeal to the behavioral specifications, the laws of behavior of that object, and relate the behavior to these specifications. Another kind of explanation is structural: show that the behavior is a result of the behavioral specifications of its components and the way they are connected. In order to produce the latter kind of explanation, the object has to be decomposed into its components, their behaviors composed and abstracted to correspond to the set of behaviors to be explained.
- *Functions.* Functions are defined in terms of desired state changes in the objects in the environment. Artifacts are designed such that they cause the desired state changes in the objects in the environment, i.e., the task of the designer is to compose components into an artifact such that the behavior of the latter causes the desired state changes in the world.

Specific contributions in this area and their benefits include:

- A formal framework for representing objects, their causal properties and their interaction. This framework can be used to build up a device simulation facility that uses the representations in the object library.
- A formal definition of function that does not make any reference to any aspect of implementation. This is in contrast to almost all current definitions that define function in terms of some aspect of the implemented artifact. Our definition enables the function of a device to be defined before the device is actually designed, and also enables retrieval by function from component libraries.

Appendix 10 describes the results in some detail.

3.2.2. Specifying Functional Requirements

Software design, like general system design, depends on determining and stating system requirements. Requirements specify the required behavior of the system in an environment, involving domain and system objects. Typical requirement artifacts are informal documents and scenario descriptions. Typical domain models are informal or semi-formal data models.

We used Functional Representation (FR) to capture and formalize understanding of system requirements and objects. This is understanding that may be informal in current requirements artifacts, or which may be present only in the minds of requirements engineers.

For example, requirements engineers know much more about an automated teller machine (ATM) and its user interactions than is formalized in requirements documents.

Existing requirements formalisms are specialized for particular aspects of the requirements. We showed how FR can represent a unified, comprehensive understanding of ATM functional requirements including: 1) the system-environment division, 2) user-system interactions and scenarios, and 3) requirement refinements through multiple abstraction levels.

FR was similarly applied to various systems engineering problems, in both hardware and software domains. This includes representation of system requirements, objects, object composition, and device libraries to support design and analysis.

Specific contributions and accomplishments in this area include:

- Representing ATM functional requirements, including environment, system, and user, refined across multiple abstraction levels.
- FR foundations for representing domain and system objects, including object properties, relationships, ports, compositions, behaviors, functions, and explanations.

Benefits and applications include a unified framework for systems requirements and objects, a practical means of refining and communicating more formal systems requirements, and tools which operate on such representations, e.g. to check consistency or generate explanations. Capturing system requirements more completely and formally benefits all subsequent system design and evolution.

Appendix 12 gives the technical details of how to apply FR for requirement specification.

3.3. FR as the Basis for Explaining Software Architecture

Our next context is software architecture. Software architectures describe how system components interact and behave. They are specified using architecture description languages (ADL's) which give the architecture's components, and their connections, interactions and behaviors. ADL's are valuable in software design and evolution. However their value is limited because they specify architectures without reference to designers' intentions.

We used Functional Representation to capture design intentions and their implementation in an architecture. Typical design intentions involve implementing abstract design goals with particular architectural structure and rationale. This is understanding which designers have, but which is typically lost because it is not recorded. FR-Rapide is our technology for FR representation of intentions implemented in the Rapide executable ADL.

An example FR-Rapide representation captures how part of the Two-Phase Commit protocol is implemented in a Rapide architecture. It incorporates understanding in domains such as transaction processing, the X/Open standard, concurrent computing, and distributed computing.

Specific contributions and accomplishments in this area include:

- An approach and examples for representing how intentions are implemented in software architectures using multiple abstraction levels and domains.
- A prototype that answers questions and generates explanations of the architecture.

Benefits and applications include a practical means of recording and communicating architecture intentions and their implementation, and tools which deliver and exploit represented understanding, e.g. for applications such as browsing, documentation, debugging, simulation, design verification, and rationale capture. Representing architecture understanding benefits many evolution tasks using architecture and ADL's.

Appendix 13 gives detailed technical description of this work.

3.4. FR Applications for Legacy Software

Finally, our work addresses the vast quantity of existing real-world software -- legacy software. Legacy code is typically procedural, written in languages like Cobol and Fortran, and not amenable to complete and correct formal specification. The legacy software artifacts considered are primarily source programs.

Much software engineering consists of understanding legacy code, and performing maintenance and evolution based on this understanding. Most existing representations of such understanding are based on shallow syntactic understanding, e.g. a flow graph, or are informal, e.g. documentation. We used Functional Representation to capture and formalize deeper understanding of how abstract functionality is implemented in existing code. This includes abstract views and explanation hierarchies for particular intentions, automatic explanation, functional components, alternative implementations, and reengineering processes.

This work was conducted in the specific context of human and automatic program understanding and reengineering. For example, we showed how FR captures understanding of how "read-process loop" functionality is implemented in the PAYDAY program (see Appendix 11) and other example programs, and understanding of how it is re-implemented when reengineering such code. In the case of human reengineering, this shows FR's generality for capturing understandings needed in a complex task. In the automatic case, this demonstrates FR's advantages for representing understanding and knowledge used in understanding and reengineering tools.

FR-UNPROG is the technology for FR representation of understanding in the UNPROG automatic program understanding and reengineering system. FR-Plans show how FR subsumes and enhances various formulations of programming plans, including plans with functional constraints and UNPROG plans.

Specific contributions and accomplishments representing understanding of legacy code include:

- FR-UNPROG representation of original and reengineered understanding in PAYDAY and other example programs.

- Examples of answering questions and generating explanations from FR-UNPROG representations, demonstrating how understanding captured by FR can be exploited in many tools and tasks.

Benefits and applications include capturing, formalizing and communicating previously informal understanding, and tools and systems that exploit such captured understanding. Appendix 11 contains technical details on the progress we have made in this problem.

3. 5. Summary of Results and Benefits

We believe that the project made considerable progress in both of the two related goals: the development of a component-based software composition technology and exploration of representations and use of software understanding in various software engineering contexts.

A number of foundational issues were explored in software component research. How algorithms can be recast to make them into reusable components, how iterators should be abstracted and encapsulated, why and how abstraction relations are needed to verify abstract data type representations, and relations between software components are some of the issues we investigated. We have proposed that a robust software component technology requires that the components be designed and composed following a discipline that permits modular reasoning. We have developed arguments regarding why reverse engineering of most legacy code is unlikely to be promising – it is costly to understand legacy code sufficiently well to permit changes to be made safely and that, unless reengineering adopts the kind of component design and composition that we advocate, comprehensible reengineering will be unattainable.

On the software comprehension side, we have shown the utility of our models of comprehension for a number of software engineering tasks. Here are examples of answers that can be given using our representations, in each context:

In Component-Based Software:

Q: What do QueueIterator components do?

A: Queue_Iterator components produce successive items in a queue using operations Start_Iterator, Finish_Iterator, Get_Next_Item and Is_Empty. The behavior and interfaces of these operations are completely described by a specification based on a mathematical model. Component code is guaranteed to implement this specification, regardless of the programs which use and are used by this component.

In Requirements Engineering:

Q: What are the functional requirements that enable a customer to withdraw cash in the environment of a bank ATM?

A: The environment must support the customer function of increasing his cash and decreasing his balance by $w, subject to certain conditions such as the environment's withdrawal limit. This requirement can be further decomposed into a plausible sequence of state transitions and subgoals to satisfy the conditions.

In Software Architecture:

Q: How does the Rapide X/Open architecture ensure transaction consistency under the Two-Phase Commit protocol?

A: The Transaction Manager decides whether the transaction is safe, then tells the Resource Managers whether to finalize or rollback the transaction. This Poll-Decide functionality is implemented by state changes controlled by the functions Poll-Decide-ok, Poll-Decide-error, Commit and Rollback. The implementation of each of these functions can be explained in more detailed models etc., leading ultimately to the distributed, concurrent Rapide code.
In Legacy Software:

Q: What does legacy program part Input_Data_2 do?

A: It implements a "read-process loop" described by a specification for a sequence of reading and processing data, given certain conditions. The implementation of Read, Process, and Termination is described using other functions and program parts etc., based ultimately on the Fortran code.

While our main results are representations that capture such software understanding, we also investigated their consequences and uses. In each context, it is easy to see that captured understanding is useful for:

- Formalizing and communicating understanding for human use
- Enabling more powerful software engineering tools and environments.

In many cases, we have shown that the needed understanding can be captured in our representations. Where this is information that has not previously been represented, except perhaps in natural language, there are obvious advantages in creating deeper, more formal, and more machine-accessible descriptions of software artifacts. In other cases, we have shown technical benefits of our approaches over existing representations.

Our work also has implications beyond the results in each software engineering context. Taken together, it gives a broad picture of the interactions between software understanding, design, understandability, tools, and information use in software evolution.

Appendices

We are enclosing the following papers as appendices to the final report. They contain technical details of the results summarized in previous pages.

1. Weide, B.W., Edwards, S.H., Harms, D.E., and Lamb, D.A., "Design and Specification of Iterators Using the Swapping Paradigm," *IEEE Transactions on Software Engineering* 20, 8 (August 1994), 631-643.

2. Weide, B.W., Ogden, W.F., and Sitaraman, M., "Recasting Algorithms to Encourage Reuse," *IEEE Software* 11, 5 (September 1994), 80-88.

3. Zweben, S.H., Edwards, S.H., Weide, B.W., and Hollingsworth, J.E., "The Effects of Layering and Encapsulation on Software Development Cost and Quality," *IEEE Transactions on Software Engineering* 21, 3 (March 1995), 200-208.

4. Weide, B.W., Edwards, S.H., Heym, W.D., Long, T.J., and Ogden, W.F., "Characterizing Observability and Controllability of Software Components," *Proceedings 4th International Conference on Software Reuse*, IEEE, Orlando, FL, April 1996, 62-71.

5. Edwards, S.H., "Representation Inheritance: A Safe Form of 'White-Box' Code Inheritance," *IEEE Transactions on Software Engineering*, 23, 2 (February 1997), 83-92.

6. Sitaraman, M., Weide, B.W., and Ogden, W.F., "On the Practical Need for Abstraction Relations to Verify Abstract Data Type Representations," *IEEE Transactions on Software Engineering*, 23, 3 (March 1997), 157-170.

7. Edwards, S.H., Gibson, D.S., Weide, B.W., and Zhupanov, S., "Software Component Relationships," *Proceedings 8th Annual Workshop on Software Reuse*, Columbus, OH, March 1997.

8. Weide, B.W., Heym, W.D., and Hollingsworth, J.E., "Reverse Engineering of Legacy Code Exposed," *Proceedings 17th International Conference on Software Engineering*, ACM, Seattle, WA, April 1995, 327-331.

9. Edwards, S. H., Modeling Modular Software Structure for Human Understanding, Technical Report, The Ohio State University, Department of Computer & Information Science, 1997.

10. B. Chandrasekaran and J. R. Josephson, Representing Function as Effect: Assigning Functions to Objects in Context and Out, AAAI-96 Workshop on Modeling and Reasoning with Function, August, 1996, Portland, OR. Also available as technical report, LAIR, CIS Department, The Ohio State University, 1996.

11. Hartman, J. and Chandrasekaran, B., Functional Representation and Understanding of Software: Technology and Application. *Proc. 5th Annual Dual Use Technologies and Applications Conference*, Rome Lab, May 1995.

12. B. Chandrasekaran and H. Kaindl, Representing Functional Requirements and User-System Interactions, *AAAI-96 Workshop on Modeling and Reasoning with Function*, August, 1996, Portland, OR. Also available as technical report, LAIR, CIS Department, The Ohio State University, 1996.

13. Hartman, J. and Chandrasekaran, B. Functional Representation of Executable Software Architectures, Technical Report, Laboratory for AI Research, CIS Department, The Ohio State University, Dec 1995.

Design and Specification of Iterators Using the Swapping Paradigm

Bruce W. Weide, *Member, IEEE*, Stephen H. Edwards, Douglas E. Harms, *Member, IEEE*, and David Alex Lamb, *Senior Member, IEEE*

***Abstract*—How should iterators be abstracted and encapsulated in modern imperative languages? We consider the combined impact of several factors on this question: the need for a common interface model for user defined iterator abstractions, the importance of formal methods in specifying such a model, and problems involved in modular correctness proofs of iterator implementations and clients. A series of iterator designs illustrates the advantages of the swapping paradigm over the traditional copying paradigm. Specifically, swapping based designs admit more efficient implementations while offering relatively straightforward formal specifications and the potential for modular reasoning about program behavior. The final proposed design schema is a common interface model for an iterator for any generic collection.**

***Index Terms*—Common interface model, formal specification, iterator, modular reasoning, program verification, proof of correctness, swapping**

I. Introduction

An *iterator* is an abstraction that supports sequential access to the individual items of a collection, without modifying the collection. Although some "academic" languages (most notably Alphard [16] and CLU [13]) include special language constructs for iterators, and others have been proposed [3], the most widely used modern imperative languages, such as Ada and C++, offer no special support for iterators. In these languages, iterators must be designed and encapsulated using the same mechanisms that are used for other user-defined abstractions: types, procedures, and packages/classes/modules. This paper discusses why previously published iterator designs are unsatisfactory in several respects, and considers the combined impact of several recent advances on the potential for improvement.

One such development is the proposal by Harms and Weide [9], [19] that swapping should replace copying as the primary data movement mechanism in imperative programs. In the *swapping style* of programming, the usual assignment operator, :=, disappears. (Of course, copying still can be achieved by calling a procedure to do it.) The universal method of data movement becomes the swap operator :=:, which exchanges the values of its two operands. This subtle change leads to several advantages for designing and implementing generic reusable software components, including improved efficiency and simplified modular reasoning about program behavior. The swapping paradigm is especially valuable when dealing with potentially large and complex data structures that represent collections of items—just the situation in which iterators are normally used.

In other recent work, Edwards [4] proposes that the swapping paradigm might be applied to the design and implementation of iterators. He also addresses a serious problem facing software component designers, i.e., developing interface models that simplify component composition. Tracz [18] discusses an example involving what Edwards [5], [6] notices is an iterator. Edwards defines a *common interface model* informally (see [7] for a formal treatment) as a convention, shared by designers of piece-part families and their potential clients, for how the plugs and sockets of plug-compatible software components are supposed to work. It includes not only parameter profiles of operations but also a shared understanding of the abstract behavior of those operations.

A third recent development is the development of formal trace specifications for iterators by Lamb [12] and by Pearce and Lamb [15]. These papers clearly explain the need for, and difficulties in, formal specification of iterators. Two related aspects of this issue that must be faced when defining a common interface model are, How should the abstract behavior of an iterator be designed so that all relevant features can be formally specified, and how can we use this specification to reason about program behavior? Especially in a component-based system, this reasoning must be modular; i.e., it must be possible to reason about the correctness of the iterator implementation independently of each client program, and vice versa. The crucial importance of, and difficulties with, modular verification of realistically large software systems in modern languages with data abstraction are noted by Ernst *et al.* [8] and Hollingsworth [10], among others.

Previous work on iteration over the elements of a composite data structure, summarized nicely by Bishop [1], has not considered together efficiency with respect to copying, the need for formal specification of a common interface model,

Manuscript received August 12, 1992; revised March 1994. The work of the first two authors was supported by the National Science Foundation under Grants CCR-9111892 and CCR-9311702, and by the U.S. Department of Defense Advanced Research Projects Agency (ARPA) under ARPA Contract F30602-93-C-0243, monitored by the U.S. Air Force Material Command, Rome Laboratories, ARPA Order A714. Recommended by M. Moriconi.

B. W. Weide and S. H. Edwards are with the Department of Computer and Information Science, Ohio State University, Columbus, OH 43210 USA; e-mail: weide@cis.ohio-state.edu, edwards@cis.ohio-state.edu.

D. E. Harms is with the Department of Mathematics and Computer Science, Muskingum College, New Concord, OH 43762 USA; e-mail: harms @muskingum.edu.

D. A. Lamb is with the Department of Computing and Information Science, Queen's University, Kingston, ON K7L 3N6 Canada; e-mail: dalamb@qucis.queensu.ca.

IEEE Log Number 9403568.

and the importance of modular reasoning about correctness in the design of iterators. This paper therefore has the following related objectives:

1) To show how to design an iterator in the swapping paradigm, which permits a most-efficient implementation, i.e., one that does not copy either items of the collection or the collection's representation data structure;
2) To give an abstract model-oriented specification of an iterator for a particular abstract collection of items, so the iterator's abstract interface is clearly and unambiguously defined;
3) To explain how this specification supports modular reasoning about the behavior of the iterator implementation and its clients, and modular verification of programs involving iterators; and
4) To demonstrate how the design can be generalized to lead to similar iterators for any abstract collection, thereby promoting composability of components.

This paper is, in effect, a proposal for a common interface model for a large class of iterators. A superficial examination of this model suggests that it is not much different from previously published iterators. In fact, however, our designs resemble others, primarily in having similar names for the operations. The *behavior* of these operations—both in functionality and performance—is subtly but importantly different.

Section II begins with a review of past work on iterators and notes the problems with previous designs. We also review the swapping paradigm and the RESOLVE notation for formal specification, and introduce a simple example that forms the basis for development of a simple iterator: a first-in, first-out (FIFO) queue abstract data type (ADT). Section III explains, step-by-step, how to arrive at the design of an acceptable swapping-style iterator for this ADT. It addresses objectives 1)–3) above for each candidate design along the way. Finally, Section IV discusses variations and extensions, and shows how the method used for the simple FIFO queue example can be generalized to a schema for specifying iterators for arbitrary collection types. All iterators designed using these principles share a common interface model, which can serve as the basis for interfaces exported by Ada generic packages and C++ class templates, among others. Example code for two typical client operations is provided in the Appendix.

II. Background

This section discusses the features required of an acceptable iterator design, the rationale for limiting the discussion to user-defined abstractions (as opposed to built-in language constructs that support iterators), relevant details of the swapping paradigm, and our approach to, and notation for, formal specification. Throughout the discussion, we refer to the client (respectively, "client program" or "client code") and to the implementer (respectively, "implementation"). The former is the programmer (respectively, program) that uses the abstract iterator concept. The latter is the programmer (respectively, program) that realizes the iterator abstraction in the form of an executable code.

A. Iterators

The simplest kind of iterator permits a client program to examine (i.e., to execute some piece of code for) each of the items of a collection without modifying the collection as a side effect of iterating over it. The items are presented to the client in some order that is based on the collection abstraction. Examples include enumerating and accumulating information about the items in a set, printing all the items in a tree, and copying a FIFO queue. There is no natural order for iterating over the elements of a set (any order will do), but there are several useful presentation orders for trees and an obvious natural order for a FIFO queue.

There are various more complex iterators and possible uses for them. For example, we might wish to be able to exit early from an iteration based on satisfaction of some condition, to have some control over the order of iteration or to leave it entirely unspecified and up to the implementer's discretion; or we might wish to change the original collection or its items while iterating over it. We begin by considering the simplest case described above, and discuss more complex cases in Section IV. A review of past work suggests that there are two subtle aspects of even the simplest iterators.

1) *Correctness:* It should not be permissible for a (correct) client program to iterate over a collection while interleaved operations on that collection might be changing it. We call this property *noninterference*.
2) *Efficiency:* It should be possible for a client program to iterate over a collection without copying the data structure that represents the collection and without copying the individual items in the collection.[1]

Correctness: Recognition of the relationship between noninterference and the modular verification of correctness dates back to attempts to verify Alphard programs involving iterators [16]. Programmers using one of Alphard's iterator constructs are advised to consider noninterference to be a restriction on its use, but no formal proof obligation is raised during verification. Proof rules should permit local verification of an implementation and its client programs, but this cannot be achieved without an assurance of noninterference, either through restriction by language syntax or by the presence of a noninterference proof obligation. Alphard, like other languages with iterator constructs, offers neither.

In an attempt to deal with noninterference in user-defined iterator abstractions, Booch [2] and Bishop [1] suggest classifying iterators into two categories, which Booch calls active and passive. An active iterator is a *module* that exports an iterator type and associated operations and permits a client to build iteration loops with standard control constructs, e.g., while loops. The main difficulty with this approach is that such a loop body may also contain calls to operations that manipulate the collection over which iteration is being done; this is precisely the problem with Alphard's and other language-supplied iterator constructs. By contrast, a passive iterator effectively encapsulates the iteration loop in a single

[1] In the special case that copying a collection is the purpose of iterating over it, all copying should take place in the client code that is executed for each item. Copying should not be inherent in the iterator itself.

procedure, which is parameterized by an action that would be the loop body in an iteration using an active iterator. The argument is that in this case there is no (obvious) way for a client to interleave operations that change the collection with those iterating over it, because the latter are encapsulated in the passive iterator procedure.

Unfortunately, passive iterators suffer from their own serious problems, discussed in detail by many authors [1], [2], [4]. From the standpoint of reusability, they are far less flexible than active iterators. For example, a client can iterate simultaneously over multiple collections with an active iterator (see the appendix), but not with a passive one. In the face of formal specification and the need for modular verification, the nature of the action procedure's effects and side effects must be formally specified and proofs modularized. It is not clear how to do this. Moreover, a client still can violate noninterference by, for instance, declaring a collection to be global to the iterator's action procedure and interfering with the iteration by manipulating that collection surreptitiously. The *coup de grâce* for passive iterators from the standpoint of reuse is the observation that an implementation of a passive iterator can be layered easily on top of an active one, but not vice versa.

Therefore, we follow the above-cited papers in concentrating on designs for active iterators. However, we insist that clients observe the noninterference property and be modularly verifiable, which necessarily makes our designs different from previous ones. That is, like Lamb [12], we write our formal specification so that noninterference *must* be observed by a correct client program. A proof obligation involving noninterference is raised in the client that can and must be discharged in a provably correct client program.

By contrast, Booch [2] points out that his iterator designs are relatively unprotected from client abuse. Indeed, nothing but self-discipline prevents a client from altering a collection during iteration over it. The same is true for Bishop's designs [1]. Several methods for repairing this shortcoming are proposed by Edwards [4]; but like Booch and Bishop, he does not deal explicitly with formal specification or the need for a framework for modular verification. These objectives drive many of our design decisions and account for the differences between Edwards's designs and the ones we propose here.

Efficiency: Although noninterference has long been seen as a problem with iterators, Edwards [4] was the first to recognize the inefficiency inherent in both published iterator abstractions and language constructs. All previously published designs for iterators (i.e., those before Edwards's papers [4]–[6]) include a function called, e.g., Value_Of. This returns to the client a *copy* of the next item from the collection.

The execution-time cost of such copying is troubling if the representations of the items in the collection are themselves large, complex data structures. As noted by Harms and Weide [9], the typical method of avoiding this expense—copying only a reference (pointer) to an item, as with the designs recommended by Booch [2] and Bishop [1]—creates even more serious problems from the standpoint of our objectives. It significantly complicates formal specification and, practically speaking, thwarts modular verification [8], [10]. This formal-proof difficulty has practical consequences: It means that human understanding of, and informal reasoning about, program behavior is much harder than it should be. Replacing copying by swapping is both efficient and amenable to tractable formal specification and modular proof rules, and hence to easier understanding of program behavior. This is the reason why we prefer the swapping paradigm for our designs.

Another efficiency issue is noted by Edwards [4] and by Lamb [12]. Achieving optimum performance of an iterator generally requires that the implementer of an iterator have access to the underlying representation of the collection. However, this is not essential solely to obtain the required functionality of an iterator, if the operations on the collection abstraction are sufficiently powerful [4], [9], [19].

B. Language Features and User-Defined Iterator Abstractions

Alphard [16] and CLU [13] have built-in iterator constructs, and Cameron [3] proposes some elegant variations. Here we concentrate on designing iterators as user-defined abstractions in languages that do not include special constructs to support iterators, and we do not further consider possible language support for our designs. There are three reasons for this. First, the practical successors to Alphard and CLU (e.g., Ada and C++) simply do not support iterators directly, so there is clearly a need for a design approach that does not rely on special language support. Second, even with language support, one needs to define formally a common interface model for iterators if a high degree of composability of software components is to be expected [5], [6]. Finally, none of the proposed language mechanisms satisfactorily addresses the problem of noninterference and the need for modular reasoning about program behavior, or the inefficiency of copying.

C. The Swapping Paradigm

The swapping style of software design [9], [19] differs from the conventional copying style in using swapping (and the swap operator :=:) to replace copying (and the standard := operator). It is based on two observations about generic modules, e.g., Ada generic packages. First, items whose types are parameters to generic modules might have large data structures as their concrete representations. These items therefore might be expensive to copy. Second, an attempt to overcome the cost of copying the abstract values of such items by copying references to them inevitably leads to difficulties in establishing program correctness by modular reasoning. This in turn frustrates both the clients of an abstraction and maintainers of its implementations. Therefore, it is advantageous to design the abstract interface of a generic component so that an implementation can achieve data movement by swapping (exchanging) the abstract values of any two variables of the same type, rather than by copying abstract values (destroying old values and duplicating new ones) or by copying references to abstract values.

Harms and Weide [9], [19] and Hollingsworth [10] propose detailed principles to help designers create generic reusable software components in the swapping style. For example, consider the operations on collection types such as a Queue of Items. Insertion operations such as Enqueue should permit

```
concept Queue_Template
   context
      parametric context
         type Item
   interface
      type Queue is modeled by string of math[Item]
         exemplar q
         initialization
            ensures q = empty_string
      operation Enqueue (
            alters q: Queue
            consumes x: Item)
         ensures    q = #q * <#x>
      operation Dequeue (
            alters q: Queue
            produces x: Item)
         requires   q /= empty_string
         ensures    <x> * q = #q
      operation Is_Empty (
            preserves q: Queue): Boolean
         ensures    Is_Empty  iff  q = empty_string
end Queue_Template
```

Fig. 1. FIFO queue specification.

implementations that swap Items into the structure. Inspection or removal operations such as Dequeue should permit implementations that swap Items out.

A particularly instructive example is an Array of Items ADT. The (single) *primary* operation should take an Array, an index, and an Item, and swap the indexed element with that Item. The usual fetch and store become *secondary* operations using this primitive. That is, they can be implemented with an insignificant performance penalty by layering on top of the primary swap-based operation if they are really needed, and in most clients they are not [9], [19].

D. Formal Specification

The main example we use throughout the rest of this paper is a FIFO queue abstraction. The formal specification of the Queue_Template concept in a dialect of RESOLVE [9], [17], [19] is shown in Fig. 1.

A **concept** specifies a generic abstract module consisting of two parts: **context**, which spells out the information needed to complete the specification, and a description of the exported **interface**. The conceptual context of Queue_Template is provided through a generic parameter, an ADT called Item. The concept exports an ADT called Queue and primary operations to Enqueue and Dequeue Items and to test if a Queue Is_Empty. This is a model-based specification in which a Queue is modeled as a mathematical string of (the mathematical model of) Items. String theory notation includes $\langle x \rangle$, where x is an Item, which denotes the string containing the single Item x; and $a * b$, where a and b are strings, which denotes the string obtained by concatenating a and b. Initially, a variable of type Queue is empty; i.e., its model is the empty string, denoted by **empty_string**.

The notation used in **ensures** clauses (postconditions) is that a variable stands for the value of its mathematical model at the conclusion of the operation; the variable prefixed with # (pronounced "old") stands for the value of the variable's mathematical model at the start of that operation. The # prefix is not needed or used in **requires** clauses (preconditions), where all variables denote values at the start of the operation.

The parameters' *modes* are used to simplify specification, and have nothing to do with the mechanism for passing parameters [9]. Mode **alters** means the argument replacing this formal parameter may be changed as a result of the call; how it is changed is stated explicitly in the postcondition, which generally relates that variable's new value to its old value and to the values of other formal parameters. Mode **preserves** means the argument's value at the conclusion of the operation is the same as it is at the start of the operation. For example, in operation Is_Empty, there is no need to say explicitly in the postcondition q = #q. Mode **consumes** means the argument's value is changed to an initial value for its type. For example, consuming a variable of type Queue would make it equal **empty_string**, while consuming an Integer would make it 0 (assuming the initial value for Integers is 0). Finally, mode **produces** means that the argument's value may be changed by the call, but its value at the beginning of the call has no influence on the operation's behavior.

Lamb [12] and Pearce and Lamb [15] use trace specifications for iterators. In this paper, we use model-oriented specifications like the one above. Model-oriented specifications seem well suited to designs based on swapping, have seen relatively widespread use in practice (e.g., Larch and *Z*), and are rather easily understood, even by those not intimately familiar with the wide variety of formal specification techniques currently in use [17], [20]. They also have been used in proof systems for modular verification of implementations and clients [8].

At the risk of seeming to apologize for writing formal specifications, we note in advance that the formal specification of the final iterator design we propose is not as short or as simple as we might have hoped. We believe this is due to the moderately complex behavior that the specification describes, inherent in iterators, and not to a serious shortcoming with either the specification notation or with our choice of how to specify iterators in that notation. We know of no comparably complex behavior specified in any formal way that does not *look* at least as imposing. The question arises, though, whether real programmers can be expected to understand such a specification, and, if not, what value it has. Others already have answered this somewhat loaded question [17], [20]. But we would add that the importance of programmer understanding of formal specifications only underscores the need for a common interface model for iterators that, once understood after, say, a careful reading of this paper, leads to rapid understanding of an entire class of structurally similar specifications [6]. We also note that even if most client programmers could understand iterators from only derived metaphorical descriptions and examples and could not read the formalism itself, then a formal specification still would serve an important role as the legal contract between implementer and client against which formal verification could be performed by experts or mechanical provers.

III. Development of an Iterator for a Queue

The goal of this section is to develop a design approach that applies to iterators for any type of collection of any type of item. We create an iterator for the generic Queue type of

ection II, then generalize in Section IV. The presentation in his section is incremental. In each step, we present a proposed lesign of the iterator and sample client code that uses it, then liscuss it, critique it, and propose a new design, until the inal design achieves the stated objectives. The development proceeds as follows:

- *Design #1:* Attack problem (1) from Section II-A, i.e., noninterference and modular verification of correctness. We define a companion type Iterator for type Queue with operations that support iteration over a Queue. The idea of this step is to make noninterference a nonissue and thereby permit modular correctness proofs. The chief problem with this design is that it is based on the copying paradigm and therefore is inherently inefficient. In fact, Design #1 might look like a straw man to some readers; after all, no one really designs iterators this way. But that is precisely the point: To enforce noninterference and achieve modularity of correctness proofs, designs based on the copying paradigm must sacrifice efficiency. Other real iterator designs attempt to achieve some degree of efficiency *at the expense of* assured noninterference and proof modularity. Design #1 illustrates that the trade-off might be made in the other direction. It also serves as the basis for better designs to follow.
- *Design #2:* Attack problem (2) from Section II-A, i.e., efficiency with respect to copying. We revise Design #1 to use swapping. The purpose of this step is to permit an implementation of an iterator that still demands noninterference and supports modular verification, yet does not need to copy either the data structure that represents the Queue or any of the Items in it. The main problems with Design #2 are that it is cumbersome to write a loop invariant to demonstrate the correctness of a typical client program, and that some swapping-paradigm principles still are not completely observed.
- *Design #3:* Add some abstract state information to the model of the Iterator type to remedy the verification problem above, and change the operations slightly to take advantage of it. The purpose of this step is to facilitate client correctness proofs and to achieve closer adherence to swapping paradigm design principles. This design achieves all the stated objectives. A generalization that handles arbitrary collections and various extensions is presented in Section IV.

A. Design #1

First, we define a companion type Iterator for the type Queue. This new type has its own operations that support iteration over a Queue. Typical client code involves two steps: Transfer the Queue value into an Iterator variable; then iterate over that variable, not over the original Queue.

An appropriate mathematical model of an Iterator is (like a Queue) a string of Items.[2] This string records the order in which the Items are to be processed during iteration. Here we choose this to be the order in which the Items would be Dequeued from the original Queue. Other orderings can be specified easily by changing the postcondition of Start_Iterator, and, for some representations of type Queue, other orderings can be implemented as easily as the natural order. (See also Section IV.) The specification for Design #1 is shown in Fig. 2.

[2] An Iterator is not modeled by a Queue, because in our model-based specification framework, an ADT's model is always a mathematical object, not another program object.

```
concept Queue_Iterator_Template
   context
      global context
         Queue_Template
      parametric context
         type Item
         facility Queue_Facility is
            Queue_Template (Item)
   interface
      type Iterator is modeled by string of math[Item]
         exemplar i
         initialization
            ensures i = empty_string
      operation Start_Iterator (
            produces i: Iterator
            preserves q: Queue)
         ensures    i = q
      operation Finish_Iterator (
            consumes i: Iterator)
      operation Get_Next_Item (
            alters i: Iterator
            produces x: Item)
         requires   i /= empty_string
         ensures    <x> * i = #i
      operation Is_Empty (
            preserves i: Iterator): Boolean
         ensures    Is_Empty  iff  i = empty_string
end Queue_Iterator_Template
```

Fig. 2. Queue_Iterator Design #1.

Discussion: Design #1 involves a specification mechanism called a *facility parameter*. A **facility** is an instance of a (generic) concept. In this case, Queue_Iterator_Template is parameterized by type Item, and by an instance of Queue_Template called Queue_Facility, which exports a Queue (of Items) ADT and associated operations.

As noted earlier, we should be able to *layer* the implementation of an iterator on top of the corresponding collection abstraction, so that the new code respects the collection abstraction, and this could be done here [9], [14], [19]. However, there are potential order-of-magnitude efficiency gains if the underlying collection and the iterator are implemented together as a single program unit with shared knowledge of the collection and iterator representations. We specify such a composite concept in Fig. 3.

Queue_With_Iterator_Template is a concept that exports the combined interfaces of Queue_Template and Queue_Iterator_Template. The **local context** section in Fig. 3 simply ties down the parameters of these two generic abstractions, so the combination of interfaces is what we require from the strong typing standpoint. This is the RESOLVE mechanism for specification or interface inheritance [11]. In subsequent discussions of efficiency of iterator operations, we refer to the direct implementation of Queue_With_Iterator_Template from Fig. 3.

Here is a sample of client code for iteration using Design #1:

```
Start_Iterator(i, q)
```

```
concept Queue_With_Iterator_Template
   context
      global context
         Queue_Template
         Queue_Iterator_Template
      parametric context
         type Item
      local context
         facility Queue_Facility is
            Queue_Template (Item)
         facility Queue_Iterator_Facility is
            Queue_Iterator_Template (Item, Queue_Facility)
   interface
      re-exports
         Queue_Facility
         Queue_Iterator_Facility
end Queue_With_Iterator_Template
```

Fig. 3. Queue_With_Iterator Specification.

```
while not Is_Empty (i) do
   Get_Next_Item (i, x)
   (* code to process x *)
end while
Finish_Iterator (i)
```

It is evident from the sample code that Design #1 achieves noninterference by defining it away. The original Queue q is completely separate from the Iterator i. The Start_Iterator operation protects q from being changed during iteration. If the code in the loop body of the sample code manipulates q, there is no interference with the iteration. Similarly, changes to x in the code to process x do not influence either q or i. Therefore, it is acceptable for a client program to manipulate q inside a loop that is iterating over i, even if i was obtained from q.

Critique: Noninterference is assured here only at the cost of efficiency. Design #1 effectively forces an implementation of Start_Iterator to copy q into i. The reason is that simply copying a reference to q or references to its Items creates aliases, and hence cannot preserve the independence of the abstract values of q and i [9], [10]. It is impossible to prove that such an implementation of Queue_With_Iterator_Template is correct outside the context of a client program, because the client program might manipulate q or its Items through these aliases. The only way to create a modularly verifiable implementation for Design #1 is to copy q (including all of its Items).

However, a clever implementation of Queue_With_Iterator_Template might defer copying the data structure that represents q (but not its Items), as long as there are no calls to Enqueue or Dequeue on the original Queue q during an iteration over i. It can keep enough internal state as part of a Queue representation to recognize that in the abstract view of these operations, q supposedly has been copied into an Iterator i. It can determine whether an iteration is in progress by monitoring whether the call to Start_Iterator has been matched by a bracketing call to Finish_Iterator. If a call to Enqueue or Dequeue occurs during an iteration, the copy of q's data structure can be made at that time. Supporting this kind of implementation is the only real reason for the Finish_Iterator operation in Design #1. In the worst case, though, copying of q is still necessary.

```
concept Queue_Iterator_Template
    conceptual context
        uses
            Queue_Template
        parametric context
            type Item
            facility Queue_Facility is
                Queue_Template (Item)
    interface
        type family Iterator is modeled by (
                future: string of math[Item]
                present: math[Item]
                original: string of math[Item])
            exemplar i
            initialization
                ensures i.future = empty_string and
                        is_initial (i.present) and
                        i.original = empty_string
        operation Start_Iterator (
                produces i: Iterator
                consumes q: Queue
                produces x: Item)
            ensures     i.future = #q and
                        i.present = x and
                        i.original = #q
        operation Finish_Iterator (
                consumes i: Iterator
                produces q: Queue
                consumes x: Item)
            requires    i.present = x
            ensures     q = #i.original
        operation Get_Next_Item (
                alters i: Iterator
                alters x: Item)
            requires    i.future /= empty_string and
                        i.present = x
            ensures     <x> * i.future = #i.future and
                        i.present = x and
                        i.original = #i.original
        operation Is_Empty (
                preserves i: Iterator): Boolean
            ensures     Is_Empty iff i.future = empty_string
end Queue_Iterator_Template
```

Fig. 4. Queue_Iterator Design #2.

We again note that nearly all previously published iterator designs do not force copying of the data structure representing the collection, but they *do* force copying of its Items in the course of iterating. In such designs, the counterpart of Get_Next_Item is a function that returns a copy of the next Item in the collection. Again, a modularly verifiable implementation may not make this copy cheaply by creating an alias to the Item. These problems are intrinsic to the copying paradigm [9], [19].

B. Design #2

Design #1 can be changed to use the swapping paradigm. The reason for doing this is to permit an implementation that does not need to copy either the data structure that represents the Queue or any of the Items in it. Two key ideas make this approach workable.

The first is a change to Start_Iterator and Finish_Iterator. Start_Iterator can be modified so that an implementation can move the original Queue into the Iterator object, and the matching call to Finish_Iterator can move the Queue back. This design relieves the implementer from responsibility for copying the data structure the represents the Queue. Moving arbitrarily large data structures in this way can be accomplished in constant (uniformly bounded) time with swapping [9].

The second idea is to define Get_Next_Item so that its implementation does not need to return a copy of the Item to the client, but can swap it out. This is possible if the client is required to pass that Item back (unchanged) in the *next* call to Get_Next_Item. In this case, the implementation can simply put the Item back into the Queue data structure, and swap out the next one to return to the client. The only real hurdle is to get the boundary conditions correct, so that the first and last calls to Get_Next_Item are not special cases.

The mathematical model of an Iterator becomes an ordered triple: a string of Items (called future) serving the same purpose as the model in Design #1, a single Item (called present) that records the Item value currently held by the client, and a string of Items (called original) that records the value of the original Queue. The complete specification for Design #2 is shown in Fig. 4, where the predicate **is_initial** means that its argument has an initial value for its type.

Discussion: Below is a sample of client code for iteration using Design #2.

```
Start_Iterator (i, q, x)
while not Is_Empty (i) do
   Get_Next_Item (i, x)
   (* code to process x without
     changing i or x *)
end while
Finish_Iterator (i, q, x)
```

Why is this specification so much more complex than Design #1? How does it permit the implementer to avoid copying the Queue data structure and its Items? How can a client check the preconditions of the Get_Next_Item and Finish_Iterator operations? We answer these and other questions below by considering how to implement Queue_With_Iterator_Template.

Fig. 5 traces an example of the effects of the sample client code segment above. It shows both the abstract models of i, q, and x (to illustrate the abstract behavior), and the critical aspects of possible concrete representations for i and q (to support performance claims). In this case, q is a Queue of Integers,[3] mathematically modeled as a string of mathematical integers; strings are shown between $\langle\ \rangle$. Fig. 5 also shows a typical Queue representation, which is a record containing two fields: f points to the front node of the queue and r to the rear. The representation of an Iterator is identical, except that there is an additional field in the representation record: p points to the node whose Item is presently held by the client (if any). These concrete representations are only illustrative; others also would achieve the claimed performance.

In the top row of Fig. 5, just before execution of the sample client code begins, i and x might have any values. For example, i might have an initial value for type Iterator and x might be 17, as illustrated. The value of x before Start_Iterator is immaterial; it is just a priming value, and the specification does not say exactly what Start_Iterator (q, i, x) returns in x. But note that $i.present$ records that value; see the second row of Fig. 5. The next three rows show the situation after the three calls to Get_Next_Item that occur in the case that the original q is modeled by the three element string $\langle 9\ 6\ 90\rangle$. The value of x after the call to Finish_Iterator is 0, because the specification says that operation consumes x.

One aspect of Fig. 5 might seem mysterious: Why are there top-level pointers to the records representing an Iterator and a Queue? These pointers are not strictly necessary in order to achieve the claimed performance; swapping of Iterators and Queues still would require only constant time, even without this extra level of indirection. However, it is important for implementing swapping in a uniformly bounded time, and for code-sharing among instances of generics, as noted in [9].

In the abstract explanation of Start_Iterator, the original value of q is remembered in $i.future$, from which Items subsequently are to be dispensed to the client by Get_Next_Item. An implementation of Start_Iterator in Design #2 need not copy the original Queue data structure in order to achieve this effect. It can acquire the original value of q by swapping. Start_Iterator is designed to consume q in order to support this implementation.

On first reading, it might appear that Start_Iterator should have to copy q in order to satisfy the postcondition clause `i.original = #q`. This also is not the case, because *i.original* is part of the *abstract* state of an Iterator. There is no implication that the *concrete* representation of an Iterator must explicitly include *i.original*, and indeed none of the other operations demands that *i.original* actually be kept for correct execution, as explained below. Adding an adjunct variable (a variable that participates in proofs but not in executable code) to the Iterator representation enables us to write a formal correspondence relation between the representation and abstract values [10].

Similarly, the postcondition clause `i.present = x` in Start_Iterator means that the Item value returned to the client in x is remembered as part of the Iterator's state. But as above, this does not require copying, because $i.present$ is only part of the abstract state of an Iterator and need not be represented concretely, unless some operation's implementation calls for that; none does here.

Similarly, Get_Next_Item need not copy an Item. Its precondition `i.present = x` requires that the client pass in as x an Item equal to the one most recently returned by Start_Iterator or Get_Next_Item. The implementation can merely put this value back into the Queue data structure (in the node referenced by field p in Fig. 5) and return the next Item by swapping it out of the structure. Again, there is no need for copying, because the Item returned must be passed back in the next call to Get_Next_Item, and so on.

When iteration is completed, the client calls Finish_Iterator. This operation's precondition requires that the client give back the one outstanding Item (whose value is $i.present$), at which point the implementation has the entire data structure and all the Items in the original Queue. It simply swaps this with parameter q to achieve the stated postcondition.

A point worth noting is that no code in the client or in the implementation *checks* the clause `i.present=x` at the

[3] This makes it easy to understand the operation of the iterator, but it also makes the example too simple to illustrate the importance of not copying an Item, which might be a far more complex type than Integer! We opted for ease of understanding in choosing the example.

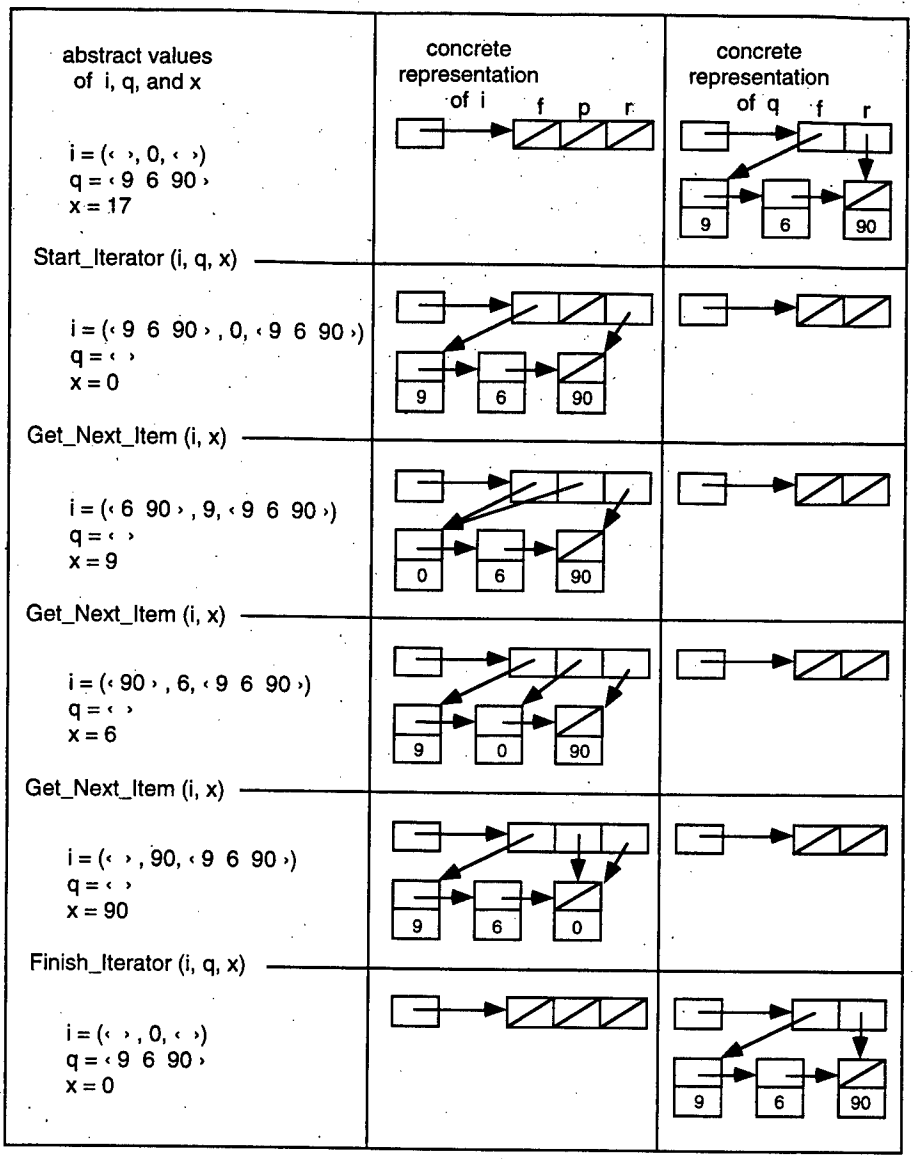


Fig. 5. Sample execution for design #2.

beginning of a call to Get_Next_Item or Finish_Iterator. In fact, because there is no operation that reveals the value of `i.present`, a client or an implementer cannot write such code without copying Items. Thus, the only means for a client to be sure that no preconditions are violated is to be able to *prove* that the code to process x does not change x.

Without the precondition on Get_Next_Item and Finish_Iterator, no such proof obligation would be raised in an arbitrary client program. Although it then might be possible to verify a particular use of the swapping-based implementation, there would be no way to separate a proof of correctness of the implementation from that of the client program. Therefore, we could not prove the correctness of this implementation in a modular fashion, and we could not declare the swapping-based implementation of Queue_Iterator_Template to be correct out of the context of a particular client. The feasibility of such a modular correctness proof was one of the primary objectives of our design.

Critique: Although Design #2 has a more complex specification than Design #1, its swapping-based implementation is straightforward and efficient. However, experience using the specification of Design #2 suggests some minor changes. Most importantly, with Design #2, it is cumbersome to show in the sample client program that the code to process x actually is executed for every item in the original Queue q. The proof relies on a loop invariant that keeps track of the Items that have been processed and relates them to the Items in $i.future$ and the original Queue. It is possible to introduce an adjunct variable for each loop to keep track of the processed Items, but it is more convenient to include support for this in the specification. This and other minor modifications are discussed in the next section.

```
concept Queue_Iterator_Template
    conceptual context
        uses
            Queue_Template
        parametric context
            type Item
            facility Queue_Facility is
                Queue_Template (Item)
    interface
        type family Iterator is modeled by (
                past: string of math[Item]
                present: math[Item]
                future: string of math[Item]
                original: string of math[Item]
                deposit: math[Item])
            exemplar i
            initialization
                ensures i.past = empty_string and
                        is_initial (i.present) and
                        i.future = empty_string and
                        i.original = empty_string and
                        is_initial (i.deposit)
        operation Start_Iterator (
                alters i: Iterator
                consumes q: Queue
                consumes x: Item)
            requires    is_initial (i)
            ensures     i.past = empty_string and
                        i.present = x and
                        i.future = #q and
                        i.original = #q and
                        i.deposit = #x
        operation Finish_Iterator (
                consumes i: Iterator
                produces q: Queue
                alters x: Item)
            requires    i.present = x
            ensures     q = #i.original and
                        x = #i.deposit
        operation Get_Next_Item (
                alters i: Iterator
                alters x: Item)
            requires    i.present = x and
                        i.future /= empty_string
            ensures     i.past = #i.past * <x> and
                        i.present = x and
                        <x> * i.future = #i.future and
                        i.original = #i.original and
                        i.deposit = #i.deposit
        operation Is_Empty (
                preserves i: Iterator): Boolean
            ensures     Is_Empty iff i.future = empty_string
end Queue_Iterator_Template
```

Fig. 6. Queue_Iterator Design #3.

C. Design #3

In Fig. 6, we add to the abstract model a field (called past) that records the Items that have been returned to the client through Get_Next_Item, and a field (called deposit) that records the priming Item that the client passed into the first call to Start_Iterator. We also add a precondition to Start_Iterator to guarantee that the Iterator i satisfies its initial condition, make Start_Iterator consume the deposited value x, modify the post condition of Finish_Iterator so that $i.deposit$ is returned in x, and reorder the Iterator model components to give a more natural reading.

The representation of an Iterator as specified in Fig. 6 might look like the representation in Fig. 5. In addition, we have to store $i.deposit$ in the concrete representation; but this is accomplished simply by swapping the value in during Start_Iterator and swapping it back out during Finish_Iterator, so there are no substantive performance implications of this change.

Discussion: Here is a sample of client code for iteration using Design #3. (See the appendix for complete client examples.)

```
Start_Iterator(i, q, x)
maintaining i.past * i.future = #i.past
            * #i.future and
            i.present = x and
            i.original = #i.original and
            i.deposit = #i.deposit
while not Is_Empty (i) do
   Get_Next_Item(i, x)
   (* code to process x without changing
     i or x *)
end while
Finish_Iterator(i, q, x)
```

In this sample code, we include the loop invariant in a **maintaining** clause, which may be considered an extra syntactic slot in the while loop construct. The notation means that at the beginning of each iteration of the loop, the concatenation of $i.past$ and $i.future$ equals their concatenation just before the loop is first encountered; that $i.present$ equals x; and that $i.original$ and $i.deposit$ equal their respective values just before the loop is first encountered.

Clearly, this invariant is true at the start of the first iteration. It is easy to show that it is true for an arbitrary iteration if and only if the code to process x does not change i or x. With the addition of the past field to an Iterator's abstract state, it also is easy to show that all Items in the original Queue q, and only those Items, are processed by the loop.

The other changes in Design #3 support a general principle of the swapping paradigm: There are advantages in simplified reasoning about program behavior and in the performance of storage management activities if temporary variables in a program act as *catalyst* variables [9]. A catalyst variable is one that is necessary to carry out a computation, but experiences no (net) change in value from the beginning to the end of the computation, or is an initial value for its type at both points. In the expected use of Queue_Iterator_Template, we want to make sure the local variables i and x are catalysts. Notice that this is not the case for Design #2; i and x might start out with any values whatsoever before Start_Iterator, and their values after Finish_Iterator might be different.

In Design #3, we therefore require that Iterator i be an initial value for its type before the call to Start_Iterator. Finish_Iterator consumes i, leaving it again as an initial value for its type. Also in Design #3, we record the priming value of x that is passed to Start_Iterator and restore that value in Finish_Iterator; thus, the name $i.deposit$, reflecting the fact that we consider the priming value to be like a security deposit that should be returned to the client upon completion of the iteration. Now both i and x act as catalyst variables.

IV. Variations and Extensions

There are several interesting variations and extensions of this approach to iterators. We briefly discuss them here, and, in the process, propose a schema for a generic Iterator_Template concept (Fig. 7) that is flexible enough to accommodate most interesting uses for iterators. This concept schema constitutes our proposal for a common interface model for iterators.

```
concept Iterator_Template
    conceptual context
        parametric context
            type Item
            type Collection
    interface
        type family Iterator is modeled by (
                past: string of math[Item]
                present: math[Item]
                future: string of math[Item]
                original:math[Collection]
                deposit: math[Item])
            exemplar i
            initialization
                ensures i.past = empty_string and
                        is_initial (i.present) and
                        i.future = empty_string and
                        is_initial (i.original) and
                        is_initial (i.deposit)
        operation Start_Iterator (
                alters i: Iterator
                consumes c: Collection
                consumes x: Item)
            requires    is_initial (i)
            ensures     i.past = empty_string and
                        i.present = x and
                        ρ (i.future, #c) and
                        i.original = #c and
                        i.deposit = #x
        operation Finish_Iterator (
                consumes i: Iterator
                produces c: Collection
                alters x: Item)
            requires    i.present = x
            ensures     c = #i.original and
                        x = #i.deposit
        operation Get_Next_Item (
                alters i: Iterator
                alters x: Item)
            requires    i.present = x and
                        i.future /= empty_string
            ensures     i.past = #i.past * <x> and
                        i.present = x and
                        <x> * i.future = #i.future and
                        i.original = #i.original and
                        i.deposit = #i.deposit
        operation Is_Empty (
                preserves i: Iterator): Boolean
            ensures     Is_Empty iff i.future = empty_string
end Iterator_Template
```

Fig. 7. Schema for a generic iterator design (with ρ free).

A. Early Exit from Iteration

A client program that exits from an iteration loop before the Iterator is empty poses no particular problem for Design #3. (See the Appendix for an example.) However, the rationale for implementing Queue_With_Iterator_Template as one module, and not layering the implementation of Queue_Iterator_Template on top of Queue_Template, is efficiency in this special case. If all the Queue_Template operations take constant time, then all the layered operations take constant time, except Finish_Iterator. In the case of an early exit from an iteration, Finish_Iterator takes time proportional to the number of Items remaining in the Iterator's future string. A direct implementation of Queue_With_Iterator_Template in which the Iterator operations have access to the underlying Queue representation (as in Fig. 5) achieves constant time performance for all operations.

B. Different Orders of Iteration and Iteration Over a Subset of All Items

It is easy to generalize the specification of Design #3 to define a schema for an Iterator type that presents the Items in a Queue to the client in a different order, and/or that iterates over just those Items that satisfy a particular condition. We define a binary (mathematical) relation:

$$\rho : \textbf{string of math}[\text{Item}] \times \textbf{string of math}[\text{Item}] \rightarrow \text{Boolean},$$

so that $\rho(s,t)$ holds whenever the order of appearance of the Items in string s is an acceptable or possible order of iteration for the desired Items in string t. We can now generalize the **ensures** clause of Start_Iterator as underlined.

```
operation Start_Iterator (
            alters i: Iterator
            consumes q: Queue
            consumes x: Item)
  requires  is_initial (i)
  ensures   i.past=empty_string and
            i.present = x and
            ρ(i.future,#q) and
            i.original= # q and
            i.deposit =# x
```

This operation specification, with ρ a free variable, should be interpreted as part of a schema for a concept, in the following sense. A specifier might use it to guide the design of different but related iterator concepts by binding ρ in any of three ways.

1) For each individual iterator concept, replace ρ by a particular relation that controls the order in which Items are to be returned by Get_Next_Item.
2) Make ρ a client-supplied parameter to the specification (like Item and Queue_Facility).
3) Make ρ an implementer-supplied parameter to the specification.

In cases (1) and (2), any realization must be further parameterized by program operations [4], [10] that satisfy certain properties involving ρ and that permit the implementer to write code that achieves the specified behavior. In case (3), the client knows only that ρ is some relation, possibly with additional mathematical properties dictated by the specifier. Here the implementer has the freedom to present the Items from the Iterator in any convenient (efficiently computed) order, and must supply a definition for ρ that characterizes the orders it might produce.

C. Other Collections

To specify Iterators for collections that are not modeled as mathematical strings, we can adapt the approach suggested above and parameterize the concept by a Collection type, as shown in Fig. 7. Again, we introduce a binary (mathematical) relation:

$$\rho: \textbf{string of math}[\text{Item}] \times \textbf{math}[\text{Collection}] \rightarrow \textbf{Boolean},$$

defined so that $\rho(s,c)$ holds whenever the order of the Items in string s is an acceptable or possible order of iteration for the desired Items in Collection c.

We need a relation here, not a function. Consider a Set ADT, where the mathematical model of a program Set is a mathematical set. Then a useful implementer-supplied relation ρ would have $\rho(s, c)$ hold exactly when every Item in set c occurs exactly once in string s. There is no natural order for iterating over the elements of a Set, but we probably want to specify that the iteration should see each element exactly once. If an implementer is free to choose any order that meets this criterion, there is substantial performance flexibility.

D. Modifying a Collection (But Not Its Items)

We now consider two more advanced kinds of iterators that involve modifying the collection during iteration. There are two sorts of changes: those that restructure the collection into an equivalent form without modifying any of its Items, and those that (instead or in addition) modify the values of the Items. An example of the first kind arises if we have a Tree ADT whose nodes are labeled by Items. We might not care about the shape of a Tree, as long as an in-order traversal produces the Items in the same order, e.g., if the Tree is used as a binary search tree. A side effect of iterating over such a Tree, then, might be that it is rebalanced.

How can we specify an iterator that has such an effect? We introduce another relation below:

σ: **math**[Collection] × **string of math**[Item] × **math**[Collection] → **Boolean**,

defined so that $\sigma(i, s, f)$ holds whenever the initial Collection i, when iterated over with the order of Items in string s, is equivalent to the final Collection f. We can then generalize the **ensures** clause of Finish_Iterator from the specification in Fig. 7, as underlined, below.

```
operation Finish_Iterator (
      consumes i: Iterator
      produces c: Collection
      alters x: Item)
 requires  i.present = x
 ensures   σ(#i.original, #i.past * #i.future, c)
           and x = #i.deposit
```

Now an implementation can return in c any Collection that is equivalent to the original Collection, offering the possibility of performance flexibility or even intentional restructuring. A degenerate case of this schema, where $\sigma(i, s, f)$ holds if and only if $\rho(s, i)$ holds and $i = f$, is the schema of Fig. 7.

E. Modifying the Items in a Collection

The intuitively obvious way to change every Item in a Collection is to iterate over the Collection and change each one as it is processed. Of course, this will not work directly with the proposed design, because getting the next Item requires the client to pass back exactly the same value that it received in the previous call to Get_Next_Item. This process works similarly for Finish_Iterator.

There are two ways to address this problem. One is to iterate over the Collection and construct the modified Collection as a new object. The appendix contains example code for copying a Queue in this way; there is no modification of each Item as it is added to the new Queue, but it is easy to see how this would be done if that were the objective. The difficulty with this as a general solution is that Items cannot be modified in place. New ones must be constructed, with the associated efficiency penalty (which is possibly significant if the Items are large) that we initially argued we should like to avoid if possible.

Another approach, then, is to further generalize the Iterator_Template design to support specifying the way in which each Item is to be modified. Again, we introduce a relation that characterizes mathematically how the modified Item values must be related to the old ones:

ν: **math** [Item] × **math** [Item] → **Boolean**.

The specifier or client should define $\nu(a, b)$ to hold if and only if b is an allowable new value corresponding to the old value a. (This relation could be generalized even further to have a third argument, a string of Items, so that new Item values could depend on the values of all previously processed Items as well.)

Now we generalize the preconditions of Get_Next_Item and Finish_Iterator, replacing `i.present = x` by ν `(i.present, x)`. We also have to do two other things. The first is to add another component to the Iterator model—a string of Items perhaps called updates—and to change the postcondition of Get_Next_Item, so that this string records the (modified) Items returned to Get_Next_Item. The second is to change the relation σ from the previous subsection, so that it depends additionally on another string of Items that includes the updates, and to change the postcondition of Finish_Iterator accordingly.

Note that modifying a collection while iterating over it, though specifiable and sometimes useful, is fraught with danger. Consider iterating over a Set of Integers, squaring each one. The abstract set model of the program Set object might have fewer elements following the iteration; e.g., -2 and 2 both yield 4. Similarly, consider iterating over a Tree of Integers, squaring each one, but trying to maintain the binary search tree property. These examples illustrate that for a correct implementation, it is insufficient just to traverse the data structure representing the Set or Tree and to perform a squaring operation on each element. This is the kind of problem, both in client understanding of iteration and in implementation efficiency, that leads us to warn against modifying Items during iteration in general, even though it causes no insurmountable technical difficulties with our specification and design approach.

V. Conclusion

Previously published iterator designs are unsatisfactory along several dimensions. The iterator design developed incrementally for Queues in Section III, and generalized to a schema for arbitrary Collections in Section IV-C, addresses the deficiencies of prior approaches in the following specific ways.

- It is designed to support efficient implementations; neither the implementer nor the client needs to copy the

data structure representing the Collection, or any of the individual Items in it.
- Its abstract behavior (including the noninterference property) is formally specified.
- Its implementations and clients can be verified independently, i.e., modularly, in the sense of [8].
- It can be specified as a schema for an independent generic concept that defines an iterator abstraction for arbitrary Collections, so all iterator abstractions in a system share a common interface model.

Because of these advantages, the iterator design in Fig. 7 should be considered as a baseline proposal for a common interface model for iterator abstractions. This baseline supports sequential access to the individual Items of a Collection in various orders, but without allowing a Collection or its Items to be modified during iteration, and is robust enough to handle any container structure where such iterations are meaningful.

A final note on language issues: Our design shows, in principle, how iterators can be abstracted and encapsulated to support modular programming and modular reasoning about program behavior. But can the design be used in real programming languages? There is no technical difficulty with Ada, because a generic package may export more than one type, as an implementation of a Collection_With_Iterator_Template must. (See the *RESOLVE/Ada Discipline* [10]. The particularly interested reader also should consult [4] for detailed examples of similar iterator designs coded in Ada.)

For C++, a mismatch with the RESOLVE language model leads to minor trouble. There is a temptation to use inheritance to combine interfaces, i.e., to make Queue_With_Iterator_Template a class derived from the Queue_Template class. However, such a C++ class effectively defines just one type, not two. This leads inevitably to nontrivial differences between the abstract specification given here and even the parameter profiles of the C++ class methods. So, another solution is preferred: Make Queue_Template and Queue_Iterator_Template separate but friend classes in order to get the required efficiency of implementation of the combined interface.

Appendix
Client Examples for Design #3

Here is a sample client for Design #3, an operation to copy a queue using an iterator.

```
operation Copy (
          preserves q1: Queue
          produces q2: Queue)
    ensures q2 = q1
begin
    variable
          cleared: Queue
          i: Iterator
          x1, x2: Item

    q2 :=: cleared
    Start_Iterator (i, q1, x1)
    maintaining i.past * i.future =
                  #i.past * # i.future and
                i.present = x1 and
                i.original = #i.original
                  and
                i.deposit = #i.deposit
                  and
                q2 = i.past
    while not Is_Empty (i) do
          Get_Next_Item (i, x1)
          Copy_Item (x1, x2)
          Enqueue (q2, x2)
    end while
    Finish_Iterator (i, q1, x1)
end Copy
```

This example illustrates simultaneous iteration over two collections, and a possible early exit from an iteration loop: an operation to determine whether two Queues are equal.

```
operation Are_Equal (
          preserves q1: Queue
          preserves q2: Queue): Boolean
    ensures  Are_Equal iff q1 = q2

begin
    variable
          i1, i2: Iterator
          x1, x2: Item
          equal: Boolean

    equal := true
    Start_Iterator (i1, q1, x1)
    Start_Iterator (i2, q2, x2)
    maintaining i1.past * i1.future =
                  #i1.past * #i1.future
                  and
                i1.present = x1 and
                i1.original =
                #i1.original and
                i1.deposit= #i1.deposit)
                  and
                i2.past * i2.future =
                  #i2.past * #i2.future
                  and
                i2.present = x2 and
                i2.original =
                  #i2.original and
                i2.deposit = #i2.deposit
                  and
                equal = (i1.past =
                  i2.past)
    while equal and not Is_Empty (i1) and
     not Is_Empty (i2) do
          Get_Next_Item (i1, x1)
          Get_Next_Item (i2, x2)
          equal := Are_Equal_Items (x1,
            x2)
    end while
    if equal and (not Is_Empty (i1) or not
```

```
        Is_Empty (i2)) then
            equal := false
    end if
    Finish_Iterator (i1, q1, x1)
    Finish_Iterator (i2, q2, x2)
    return equal
end Are_Equal
```


ACKNOWLEDGMENT

We are indebted to W. Heym, J. Hollingsworth, B. Ogden, M. Sitaraman, S. Zweben, and the anonymous referees for many helpful comments.

REFERENCES

[1] J.M. Bishop, "The effect of data abstraction on loop programming techniques," *IEEE Trans. Software Eng.*, vol. 16, pp. 389–402, Apr. 1990.

[2] G. Booch, *Software Components with Ada.* Redwood City, CA: Benjamin-Cummings, 1987.

[3] R.D. Cameron, "Efficient high-level iteration with accumulators," *ACM TOPLAS*, vol. 11, pp. 194–211, Apr. 1989.

[4] S.H. Edwards, "An approach for constructing reusable software components in Ada," Tech. Rep. P-2378, Inst. for Defense Analyses, Alexandria, VA, USA, 1990.

[5] ——, "Common interface models for components are necessary to support composability," *Proc. 4th Ann. Workshop on Software Reuse*, SPC, Herndon, VA, USA, 1991.

[6] ——, "Common interface models for reusable software," *Int. J. Software Eng. Knowledge Eng.*, vol. 3, pp. 193–206, June 1993.

[7] ——, "A formal model of software subsystems," Ph.D. dissertation, Dept. of Comput. and Inform. Sci., Ohio State Univ., Columbus, OH, USA, in preparation.

[8] G.W. Ernst, R.J. Hookway, and W.F. Ogden, "Modular verification of data abstractions with shared representations," *IEEE Trans. Software Eng.*, vol. 20, pp. 288–307, Apr. 1994.

[9] D.E. Harms and B.W. Weide, "Copying and swapping: Influences on the design of reusable software components," *IEEE Trans. Software Eng.* vol. 17, pp. 424–435, May 1991.

[10] J.E. Hollingsworth, "Software component design-for reuse: A language independent discipline applied to Ada," Ph.D. dissertation, Dept. of Comput. and Inform. Sci., Ohio State Univ., Columbus, OH, USA, 1992.

[11] W.R. LaLonde, "Designing families of data types using exemplars," *ACM Trans. Programming Languages Syst.*, vol. 11, pp. 212–248, 1989.

[12] D.A. Lamb, "Specification of iterators," *IEEE Trans. Software Eng.*, vol. 16, pp. 1352–1360, Dec. 1990.

[13] B. Liskov, A. Snyder, R. Atkinson, and C. Schaffert, "Abstraction mechanisms in CLU," *CACM*, vol. 20, no. 8, pp. 564–576, Aug. 1977.

[14] S. Muralidharan and B.W. Weide, "Should data abstraction be violated to enhance software reuse?" *Proc. 8th Ann. Natl. Conf. Ada Technol.*, 1990, pp. 515–524.

[15] T.W. Pearce and D.A. Lamb, "The property vector specification of a multiset iterator," *Proc. 14th Int. ACM/IEEE Conf. Software Eng.*, 1992.

[16] M. Shaw, W.A. Wulf, and R.L. London, "Abstraction and verification in Alphard: defining and specifying iteration and generators," *CACM*, vol. 20, no. 8, pp. 553–564, Aug. 1977.

[17] M. Sitaraman, L.R. Welch, and D.E. Harms, "On specification of reusable software components," *Int. J. Software Eng. Knowledge Eng.*, vol. 3, pp. 207–229, June 1993.

[18] W. Tracz, "Parameterization: A case study," *Ada Lett.*, vol. 9, pp. 92–102, May/June 1989.

[19] B.W. Weide, W.F. Ogden, and S.H. Zweben, "Reusable software components," in M.C. Yovits, Ed., *Advances in Computers*, vol. 33. New York: Academic, 1991, pp. 1–65.

[20] J.M. Wing, "A specifier's introduction to formal methods," *Comput.*, vol. 23, pp. 8–24, Sept. 1990.

B.W. Weide (S'73–M'78) received the B.S.E.E. degree from the University of Toledo, OH, USA, and the Ph.D. degree in computer science from Carnegie Mellon University, Pittsburgh, PA, USA.

He is an Associate Professor of Computer and Information Science at Ohio State University, Columbus, OH, USA, and Codirector of the Reusable Software Research Group with Bill Ogden and Stu Zweben. His research interests include all aspects of software component engineering, especially in applying RSRG work to Ada and C++ practice.

Dr. Weide is a member of the IEEE, ACM, and CPSR.

S.H. Edwards received the B.S.E.E. degree from the California Institute of Technology and the M.S. degree in computer and information science from Ohio State University, Columbus, OH, USA.

He is a Ph.D. degree candidate in computer and information science at Ohio State University. Prior to attending Ohio State, he was a Member of Research Staff at the Institute for Defense Analyses. His research interests are in software engineering and reuse, formal models of software structure, programming languages, and information retrieval technology.

Mr. Edwards is a member of the IEEE Computer Society and ACM.

D.E. Harms (S'87–M'88) received the B.S. from Muskingum College, New Concord, OH, USA, and the M.S. and Ph.D. degrees from Ohio State University, Columbus, OH, USA.

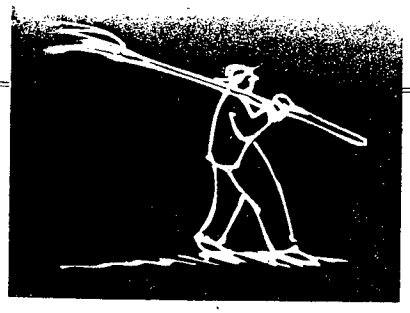
He is an Associate Professor of computer science at Muskingum College. His research interests are in software engineering (especially reuse, specification, and verification) and programming language design.

Dr. Harms is a member of the IEEE and ACM.

D.A. Lamb (S'75–M'77–SM'87) received the Ph.D. degree in computer science from Carnegie Mellon University, Pittsburgh, PA, USA, in 1983.

He is an Associate Professor of Computing and Information Science at Queen's University, Kingston, ON, Canada. His research interests include software design methods, configuration management, and formal methods in software engineering.

Dr. Lamb is a member of ACM and Sigma Xi.



Recasting Algorithms to Encourage Reuse

BRUCE W. WEIDE and WILLIAM F. OGDEN,
Ohio State University
MURALI SITARAMAN, *West Virginia University*

Instead of viewing algorithms as single large operations, the authors use a machine-oriented view to show how they can be viewed as collections of smaller objects and operations. Their approach promises more flexibility, especially in making performance trade-offs, and encourages black-box reuse. They illustrate it with a sample design of a graph algorithm.

All large software systems are built from components of some kind. A typical modern software component is a *module*, which usually encapsulates an abstract data type. The data type, in turn, hides the details of both concrete data structures and the algorithms that implement operations to manipulate the abstract data type's variables.

Reusable software components are just modules that have been carefully designed to be useful in several programs, even unanticipated ones.[1] We focus here on two types of flexibility — functional and performance — that make components reusable. We also advocate a systematic black-box style of reuse, in which designers use components without source-code modification. This contrasts to a haphazard opportunistic style in which designers scavenge old code for interesting tidbits to reshape.

We recommend black-box reuse because the real value of reused code lies in its properties, such as correctness with respect to an abstract specification. If you make even small structural or environmental changes, the confidence in these properties tends to evaporate, and with it most of the component's value.

In this article we show how to design an entire category of more flexible black-box reusable software components by applying a general design technique that "recasts" algorithms as objects. To illustrate the technique, we recast a sorting algorithm and a spanning-forest algorithm into objects.

0740-7459/94/$04.00 © 1994 IEEE

RECASTING FOR FLEXIBILITY

Conventional object-oriented design treats application-specific entities as objects and application-specific actions as operations on those objects. Many of these operations change the objects a great deal. Because they are implemented as single operations, they involve algorithms that manipulate complex data structures extensively.

The recasting technique we propose is a refinement of object-oriented design — it turns a single large-effect operation into an object by regarding it as a machine that performs the action. This effectively replaces one operation with an entire module. The module defines an abstract data type — which records the machine state — and several operations — each of which has a smaller effect. One of these smaller effect operations might supply input to the machine, for example; another might return results. This kind of design has greater *functional flexibility* — the component can be readily adapted to provide good solutions to any problem requiring its general services. A design that uses smaller effect operations does two things. First, it provides a finer grain of control. Second, it gives implementers the opportunity to offer more *performance flexibility* — they can substitute alternative implementations of an abstract component by making trade-offs among individual operations. This changes the component's performance characteristics but retains the same functional behavior.

Recasting works for two reasons:

- Component designers can organize data processing along one of two dimensions: The usual *object-structure* dimension relates items according to their explicit representation as data objects using arrays, records, lists, trees, and so on. Our recasting approach adds a *temporal* dimension, which relates items by the time they appear in a program.
- It takes advantage of the widely recognized fact that an abstract behavior specification does not prescribe *how* behavior is to be realized. In fact, module specification hides the knowledge of both how and *when* computations actually take place.

When you design a component to use large-effect operations, you are confining yourself primarily to the object-structure dimension. You miss the opportunity to use the temporal dimension as a data organizer and so preclude some potentially efficient implementations of the desired abstract behavior. Once you realize you can amortize the cost of an algorithm among several operations in the module and retain the same functionality, you gain tremendous flexibility. You can use precomputation, batch computation, deferred computation, and related data-structuring and algorithm-design techniques.[2] This gives various options to applications (like online and real-time systems) that demand that individual operations exhibit certain constrained performance profiles in addition to — or even instead of — optimal performance for an entire operation sequence.

THE MACHINE MODEL LETS DESIGNERS TUNE HOW A COMPONENT PERFORMS, NOT WHAT IT DOES.

SAMPLE DESIGN PROBLEM

To explore the nature and benefits of recasting single large-effect operations as objects, we present a traditional design problem and show how the usual design is flawed from the viewpoint of reusability. The design problem is to implement part of a circuit-layout tool: *Given an output terminal and a set of input terminals to which it must connect, determine how the terminals should be wired together in a net that minimizes total wire length.*

The key to attacking this problem is an abstract mathematical model. In this case, we can reuse well-developed ideas from the algorithms community: The required layout is a *minimum spanning tree* of an edge-weighted graph — a subset of the graph's edges that connect its vertices with a minimum total weight. The vertices are the terminals to be connected, and the edges are weighted by the lengths of wire required to connect corresponding terminal pairs.[2]

Whether you use a traditional functional design approach modified to embrace information-hiding principles[3] or a conventional object-oriented design approach, a typical solution might be

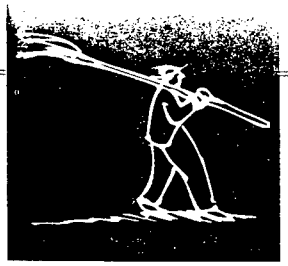
1. *Find the abstractions to be encapsulated in modules, identify their operations, and specify interface behavior.* Here you encapsulate the graph abstraction in a module and identify both operations sufficient for constructing a circuit model and implementing an operation, `Find_MST`, to compute the graph's minimum spanning tree.

2. *Implement the graph module and its associated operations.* You might use any of the many textbook graph representations.[2,4]

3. *Write a client program that uses the graph module and* `Find_MST` *to*

- *construct a graph* g *that models the portion of the circuit for which a net* t *is to be selected and*
- *find a subgraph* t *of* g *that is a minimum spanning tree of* g.

The graph module should be reusable in this and in other applications — if you carefully design it to be reused and not just to support finding minimum spanning trees. However, this design is fundamentally flawed from the standpoint of reusability and maintainability. The `Find_MST` operation is a single large-effect operation. It can be recast to make the task of finding a minimum spanning tree a separate object, offering the programmer who may want to reuse it a design that is more flexible.



Maintenance change. To illustrate, what happens when the users of this layout tool request "minor" changes after it is in the field? For example, suppose the total required wire length cannot exceed a certain bound or the output terminal's electrical features must be adjusted to handle a heavy load. This in turn might re-quire changing some terminal locations, until the net's total wire length is within the required bound. At that point, you can use the original net-selection operation to finish the job. Thus, you must now add the subtask

♦ *Determine if the total wire length of a net exceeds a given bound.*

This operation must be invoked repeatedly, with different graphs and bounds before a net can be selected; something the original code did in one invocation of the net-selection operation.

You can easily solve the bounds-checking problem by adding a step to the third part of the original solution:

♦ *Determine if the total edge weight of* t *exceeds the given bound.*

Unfortunately, this change causes users to complain of poor performance for some nets — and you find that changing the graph module or the `Find_MST` operation does not significantly improve the situation. How can you tune performance?

There is no easy solution to this problem because the decision to design `Find_MST` as a single operation has limited its functional and performance flexibility. Consequently, you must break into the `Find_MST` code to tune performance — eschewing black-box reuse and all its advantages.

Sorting algorithm. Suppose for the moment you are satisfied with the original net-selection program design. You might continue by refining the implementation of the `Find_MST` operation, which eventually should lead to something like textbook code.[2,4] Here's what might happen along the way.

CONVENTIONAL DESIGN FORCES DEVELOPERS TO 'PEEK UNDER THE COVERS' TO TUNE A COMPONENT.

First, you must choose a method for finding a graph's minimum spanning tree. The one we describe here is Kruskal's algorithm,[2, 4] a greedy algorithm that combines smaller trees joining subsets of vertices. This set of trees is called a *spanning forest.* To build it, the algorithm starts with an empty set of edges, T (a spanning forest in which each vertex is connected only to itself), looks at the edges of the graph, E, in nondecreasing order of edge weight, and adds a candidate edge to T if that edge does not form a cycle with those already in T. If the original graph is connected by E then T eventually contains a single tree, which is a minimum spanning tree. Otherwise, T eventually contains a minimum spanning forest of the original graph.

Because Kruskal's algorithm examines the edges in nondecreasing order of edge weight, it might be best to look at the problem in terms of sorting:

Given a list of items and some ordering relation on them, organize the list into nondecreasing order (where "smallest" describes the first item).

You might call a procedure from the body of `Find_MST`:

```
procedure Sort_List
  (e_list: edge_list)
```

to sort the edges in `e_list` (the edges of the graph) into nondecreasing order of edge weight.

Because Kruskal's algorithm can terminate when it discovers a minimum spanning tree, you might not have to examine many edges. This suggests a variant of sorting in which the problem is to enumerate the k smallest of n items in nondecreasing order. Unfortunately, in this case it is hard to capture this behavior in a single procedure because k is not predictable in advance — you don't know how many edges Kruskal's algorithm will need to examine before it terminates.

So you must separate the (partial) sorting problem into two phases:

1. Construct a data structure containing the set of edges that are to be examined in sorted order.

2. Incrementally deliver one edge at a time to Kruskal's algorithm, on demand, until it needs no more edges to form a minimum spanning tree.

In some textbook implementations of Kruskal's algorithm,[4] this is essentially how things work. Phase 1 consists of creating a heap data structure containing the edges, and phase 2 involves removing edges one at a time from the heap. The heap organization guarantees that the edges come out in nondecreasing order of edge weight.

This design makes it possible to have reasonable overall runtime for `Find_MST` because it lets you precompute a sorted order during phase 1, during the transition between phases 1 and 2, during phase 2, or during any of these, spreading the work around. Most important, it does not need to take as much time to get to phase 2 as it would if the algorithm sorted all the edges. So, if Kruskal's algorithm terminates before examining all the edges, the total time spent on the (partial) sorting can be substantially less than with the single Sort_List operation.

David Parnas' famous KWIC ("keyword in context") example notes the advantage of breaking up sorting into slightly smaller chunks of functionality.[3] However, to our knowledge this basic idea has neither been touted as being as general as it is nor been further developed and systematically applied to the design of reusable components. Twenty years after Parnas' paper, object-oriented component libraries still encapsulate data structures as objects and algorithms as single operations.

RECASTING SORTING

To solve the sorting variation in the `Find_MST` operation and produce a highly reusable software component, you must recast sorting as an object. Our recasting approach is based on a machine-oriented-design paradigm, in which you begin by viewing sorting as a machine that puts things of type `Item` into a sorted order. In this case, `Item` is a graph edge that you want to sort by the usual less-than-or-equal-to order on edge weights. But the module might as well be generic so it can be used with other `Item`s and other orderings.

Sorting machine data type. Imagine a sorting machine that accepts items to be sorted, one at a time, then dispenses items, one at a time, in sorted order. In many applications, you must insert all the Items before extracting the first one. There are two distinct phases: an insertion phase and an extraction phase.

Our encapsulation of a sorting machine into a module exports an abstract data type, `Sorting_Machine_State`, which records a machine state, and six operations. (Here, `m` is of type `Sorting_Machine_State` and `x` is of type `Item`).

- `Change_To_Insertion_Phase(m)`: Prepare `m` for calls to the `Insert` operation. This operation requires that `m` be in the extraction phase at the time of the call.
- `Insert(m,x)`: Insert `x` into `m`. This operation requires that `m` be in the insertion phase at the time of the call.
- `Change_To_Extraction_Phase(m)`: Prepare `m` for calls to the `Extract` operation. This operation requires that m be in the insertion phase at the time of the call.
- `Extract(m,x)`: Extract a smallest (remaining) `Item` from `m`, returning it in `x`. This operation requires that m be in the extraction phase at the time of the call.
- `Size(m)`: Return the number of `Items` in `m`.
- `Is_In_Insertion_Phase(m)`: Test if m is in the insertion phase.

Figure 1 shows the specification for this machine in Resolve.[5-7]

Intuitively, you may think of the collection of items in a sorting machine as a set, but this has two problems: First, sets have no duplicate elements, although you should be able to sort even with duplicate items. Second, sets have no intrinsic order among their elements. Using a multiset or bag (`INVENTORY_FUNCTION` in Figure 1) solves the first problem. You can address the ordering problem by specifying the `Extract` operation so that it selects, from among those items remaining, one that is smallest with respect to the desired ordering.

Functional flexibility. The `Sorting_Machine_Template` component is functionally more flexible than a single `Sort_List` operation. If you must sort all items in a collection and want a procedure like `Sort_List`, you can layer it on top of `Sorting_Machine_Template`. But if you must find only the *k* smallest items, or remove items until some condition is met, then you can stop after partial sorting.

This design has other advantages. For example, single large-effect operations such as `Sort_List` must operate on a particular data structure (it may be concrete or abstract, but it must be a particular kind in any case). In `Sort_List`, this structure is a list. If a program doesn't happen to have its data in list form, it must translate it into that form. `Sorting_Machine_Template` requires neither the source nor destination of the data to be a particular data structure or even the same kind of structure. For example, if you must get items from an input device and put them into a sorted list, you can easily layer code on `Sorting_Machine_Template` to do this.

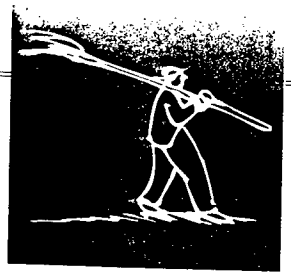
IN OUR DESIGN, NEITHER THE DATA'S SOURCE NOR ITS DESTINATION MUST BE A CERTAIN DATA STRUCTURE.

Performance flexibility. The improved performance flexibility of `Sorting_Machine_Template` over `Sort_List` comes from recognizing a key point: The abstract specification of functionality is not a prescription for *how* data structures are represented or *when* sorting actually takes place. Of course you can achieve the specified behavior by representing `Sorting_Machine_State` as a list of items and a Boolean phase flag. And you can implement the `Insert` operation by adding a new `Item` anywhere in the list; the `Change_To_Extraction_Phase` operation by toggling the phase flag; and the `Extract` operation by searching for, removing, and returning the smallest `Item` in the list.

But there are many other implementation strategies with different performance profiles:

- During each call to `Insert`, maintain the list in sorted order.
- During `Change_To_Extraction_Phase`, sort the list explicitly using any sorting algorithm.
- Represent a `Sorting_Machine_State` using a binary search tree or a heap or any other data structure, in each case facing similar choices for what each operation should do to that structure. You can precompute to any extent during each `Insert` operation; batch process during `Change_To_Extraction_Phase`; defer work as long as possible until an `Extract` operation requires it; or amortize the effort among these operations in other ways.

A good choice for a minimum-spanning-tree application is to embed heapsort so that `Change_To_Extraction_Phase` creates a heap but does



not sort the items. The fact that sorting is a two-phase operation makes this implementation possible. And some secondary operations (like finding a smallest `Item` or a *k*th smallest `Item`) run faster with this implementation than with one that sorts everything in `Change_To_Extraction_Phase`.

RECASTING FIND_MST

Returning to the original net-selection problem, note the parallels between our variation of sorting and a minimum-spanning-tree algorithm. Once you realize that you may not have to sort all items, you can profitably recast sorting as an object. The same applies to obtaining edges. If you don't need to obtain all the edges of a

```
concept Sorting_Machine_Template
  context
    global context
      Standard_Boolean_Facility
      Standard_Integer_Facility
    parametric context
      type Item
      math operation ARE_ORDERED (
          x: math[Item],
          y: math[Item]
        ): boolean
      restriction(* ARE_ORDERED is a total
          pre-ordering *)
  local context
    math subtype INVENTORY_FUNCTION is
        function from math[Item] to integer
      exemplar f
      constraint for all x: math[Item]
        (f(x) >= 0)
    math operation EMPTY_INVENTORY:
        INVENTORY_FUNCTION
      definition for all x: math[Item]
        (EMPTY_INVENTORY (x) = 0)
    math operation IS_FIRST (
        f: INVENTORY_FUNCTION
        x: math[Item]
      )
      definition f(x) > 0 and
        for all y: math[Item] where
          ARE_ORDERED (y, x) and
          not ARE_ORDERED (x, y)
          (f(y) = 0)
  interface
    type Sorting_Machine_State is modeled by
        (
      count: INVENTORY_FUNCTION
      insertion_phase: boolean
    )
    exemplar m
    initialization
      ensures m = (EMPTY_INVENTORY, true)
    operation Change_To_Insertion_Phase (
        alters m:
      Sorting_Machine_State
      )
      requires not m.insertion_phase
      ensures m = (EMPTY_INVENTORY, true)
    operation Insert (
        alters m:
      Sorting_Machine_State
        consumes x: Item
      )
      requires m.insertion_phase
      ensures differ (m.count, #m.count,
          {#x}) and
        m.count(#x) = #m.count(#x)
            + 1 and
        m.insertion_phase
    operation Change_To_Extraction_Phase (
        alters m: Sorting_Machine_State
      )
      requires m.insertion_phase
      ensures m = (#m.count, false)
    operation Extract (
        alters m:
      Sorting_Machine_State
        produces x: Item
      )
      requires m.count /= EMPTY_INVENTORY
        and not m.insertion_phase
      ensures IS_FIRST (#m.count, x) and
        differ (m.count, #m.count,
          {x}) and
        m.count(x) = #m.count(x)
          - 1 and
        not m.insertion_phase
    operation Size (
        preserves m: Sorting_Machine_State
      ): Integer
      ensures Size = sum x:math[Item]
          (m.count(x))
    operation Is_In_Insertion_Phase (
        preserves m:
      Sorting_Machine_State
      ): Boolean
      ensures Is_In_Insertion_Phase iff
```

Figure 1. Specification of a sorting-machine concept.

```
concept Spanning_Forest_Machine_Template

context

  global context

    Standard_Boolean_Facility
    Standard_Integer_Facility

  parametric context

    constant max_vertex: Integer
    restriction  max_vertex > 0

  local context

    math subtype EDGE is (
                  v1: integer
                  v2: integer
                  w: integer
      )
      exemplar   e
      constraint 1 <= e.v1 <= max_vertex and
                 1 <= e.v2 <= max_vertex and
                 e.w > 0
    math subtype GRAPH is set of EDGE

    math operation IS_MSF (
          msf: GRAPH
          g: GRAPH
      ): boolean
       definition (* true iff msf is an
          MSF of g *)
  interface
    type Spanning_Forest_Machine_State
      is modeled by (
         edges: GRAPH
         insertion_phase: boolean
      )
      exemplar    m
      initialization
        ensures    m = (empty_set, true)

  operation Change_To_Insertion_Phase (
        alters    m: Spanning_Forest_Machine_
                     State
    )
    requires   not m.insertion_phase
    ensures    m = (empty_set, true)

  operation Insert (
        alters    m:Spanning_Forest_Machine_
                     State
        consumes  v1: Integer
        consumes  v2: Integer
        consumes  w: Integer
      )
    requires    m.insertion_phase  and
                1 <= v1 <= max_vertex  and
                1 <= v2 <= max_vertex  and
                w > 0
    ensures     IS_MSF (m.edges,
                   #m.edges union
                      {(#v1, #v2, #w)})  and
                m.insertion_phase

  operation Change_To_Extraction_Phase (
        alters       m:Spanning_Forest_
                        Machine_State
      )
    requires     m.insertion_phase
    ensures      m = (#m.edges, false)

  operation Extract (
        alters       m:Spanning_Forest_
                        Machine_State
        produces     v1: Integer
        produces     v2: Integer
        produces     w: Integer
      )
    requires      m.edges /= empty_set  and
                  not m.insertion_phase
    ensures       (v1, v2, w) is in
                    #m.edges  and
                  m = (#m.edges without
                    {(v1, v2, w)}, false)

  operation Size (
        preserves    m:Spanning_Forest_
                        Machine_State
      ): Integer
    ensures          Size = m.edges

  operation Is_In_Insertion_Phase (
      preserves
m:Spanning_Forest_
                        Machine_State
      ): Boolean
    ensures  Is_In_Insertion_Phase  iff
                  m.insertion_phase
end Spanning_Forest_Machine_Template
```

Figure 2. Specification of a spanning-forest-machine concept.

minimum spanning tree, you can profitably recast this operation as an object.

Two phases. Imagine a spanning-forest machine that accepts weighted edges of a graph, one at a time, then dispenses the edges of a minimum spanning forest, one at a time. (We call this a spanning-forest machine, not a spanning-tree machine, because the original graph might not be connected by its edges. In the net-selection application the graph presumably is connected and everything will work fine. But the specification is easier, implementations are essentially the same, and the component is more reusable if the machine can find the minimum spanning forests of unconnected graphs, too.)

Should a spanning-forest machine have two phases? The same factors that influenced the design of `Sorting_Machine_Template` suggest that it should have. But there is also another reason. Given the way a minimum spanning forest is defined, it doesn't make much sense to ask for the next edge in a graph that is changing as you extract its edges. Edges previously extracted could be made erroneous as new edges are inserted. This additional reason supports the logic behind making this a two-phase machine.

How you define a spanning-forest machine is important. It is best to explain it as an "organizer" machine: The `Insert` operation promises to

```
realization Kruskal_Amortized
  for Spanning_Forest_Machine_Template
  context
    global context
      ...
    parametric context
      ...
    local context
      type Edge is record
          vertex1: Integer
          vertex2: Integer
          weight: Integer
        end record
      facility Sorting_Machine_Facility is
          Sorting_Machine_Template
            (Edge, EDGES_ARE_ORDERED)
        realized by Heapsort_Embedding (...)
      facility Coalesceable_Equivalence_
        Relation_Facility is
          Coalesceable_Equivalence_
          Relation_Template (max_vertex)
        realized by Disjoint_Set (...)
      type Spanning_Forest_Machine_
      State_Rep is record
          graph_edges: Sorting_Machine_State
          are_connected: Coalesceable_
            Equivalence_Relation
          num_spanning_edges: Integer
      end record
      ...
  interface
    type Spanning_Forest_Machine_State
      is represented by
        Spanning_Forest_Machine_State_Rep
      convention          (* rep invariant *)
      correspondence      (* representation-
        abstraction relation *)

    operation Change_To_Insertion_Phase (
          alters      m: Spanning_Forest_
                        Machine_State
        )
        new_rep: Spanning_Forest_Machine_
                 State_Rep
      begin
        m.rep :=: new_rep
      end Change_To_Insertion_Phase

    operation Insert (
          alters      m: Spanning_Forest_
                        Machine_State
          consumes v1: Integer
          consumes v2: Integer
          consumes w: Integer
        )
      begin
        if not Are_Equivalent
            (m.rep.are_connected, v1, v2)
          then
            Make_Equivalent
                (m.rep.are_connected, v1, v2)
            m.rep.num_spanning_edges :=
              m.rep.num_spanning_edges + 1
        end if
        Insert (m.rep.graph_edges, (v1, v2, w))
      end Insert

    operation Change_To_Extraction_Phase (
          alters      m: Spanning_Forest_
                        Machine_State
        )
        new_equivalence_relation:
          Coalesceable_Equivalence_Relation
      begin
        Change_To_Extraction_Phase
          (m.rep.graph_edges)
        m.rep.are_connected :=:
          new_equivalence_relation
      end Change_To_Extraction_Phase

    operation Extract (
          alters      m: Spanning_Forest_
                        Machine_State
          produces    v1: Integer
          produces    v2: Integer
          produces    w: Integer
        )
      begin
        loop
          maintaining (* loop invariant *)
          Extract (m.rep.graph_edges,
            (v1,v2,w))
          if not Are_Equivalent
              m.rep.are_connected,v1,v2)
            then
              Make_Equivalent
                (m.rep.are_connected, v1, v2)
              m.rep.num_spanning_edges :=
                m.rep.num_spanning_edges - 1
              exit
          end if
        end loop
    end Extract

    operation Size (
          preserves   m: Spanning_Forest_
                        Machine_State
        ): Integer
      begin
        return m.rep.num_spanning_edges
      end Size

    operation Is_In_Insertion_Phase (
          preserves   m: Spanning_Forest_
                        Machine_State
        ): Boolean
      begin
        return Is_In_Insertion_Phase
                  m.rep.inserting
      end Is_In_Insertion_Phase
  end Spanning_Forest_Machine_Template
```

Figure 3. An amortized-cost implementation of Spanning_Forest_Machine_Template.

keep only edges that are part of a minimum spanning forest, and the Extract operation simply removes and returns some remaining ones, as Figure 2 shows.

You can use the component in Figure 2 to solve the original net-selection problem with one Spanning_Forest_Machine_State, m:

```
while some edge of g is not
   yet inserted into m do
  let (v1, v2, w) be any edge
   of g not yet inserted
   into m
  Insert (m, v1, v2, w)
end while
Change_To_Extraction_Phase (m)
while Size (m) > 0 do
  Extract (m, v1, v2, w)
  record/report that (v1, v2,
   w) is an edge of t
end while
```

An interesting feature of this code is that you can easily change the second loop to find if a net's total wire length exceeds a given bound:

```
total_weight := 0
while (Size (m) > 0 and
   total_weight <= bound) do
  Extract (m, v1, v2, w)
  total_weight :=
   total_weight + w
end while
exceeds_bound :=
  (total_weight > bound)
```

This change by itself is not particularly easier or harder to make than for the original design. But other ramifications of the new design are significant. Now you can tune performance without "peeking under the covers" into Kruskal's algorithm. All you need to do is select an implementation that amortizes the costs of Spanning_Forest_Machine_Template so that the Insert and Change_To_Extraction_Phase operations don't actually compute a minimum spanning forest — it's almost all done in the Extract operation.

Amortizing costs. Model-based formal specifications do not favor any particular implementation and certainly do not limit you to a single implementation. So, despite this specification's claim that Insert keeps only minimum-spanning-forest edges, you are free to amortize the cost of finding a minimum spanning forest among Insert, Change_To_Extraction_Phase, and Extract in any way that makes sense.

For example, you could choose to do all the interesting work during Change_To_Extraction_Phase. To do so, build a graph from the inserted edges, use Kruskal's algorithm to find a minimum spanning forest, and save the spanning-forest edges in a list (for example) from which they can be dispensed during subsequent Extract operations. This gives the same performance as the original solution, and it also means that you pay for finding a minimum spanning forest even if you don't need to extract all the edges — precisely the performance problem raised by the maintenance example.

MACHINE-ORIENTED DESIGN IS EASIER FOR CLIENTS TO UNDERSTAND.

Instead, your implementation could defer computation until the Extract operation, as Figure 3 shows. Represent a Spanning_Forest_Machine_State (in part) with a Sorting_Machine_State whose Item type is a record that contains two vertices and a weight for a single edge. The EDGES_ARE_ORDERED relation is less-than-or-equal-to on the weight field. Then call the Insert operation on the Sorting_Machine_State to add the new edge and call the Change_To_Extraction_Phase operation on the Sorting_Machine_State to change the phase of the Spanning_Forest_Machine_State. Finally, keep calling the Extract operation on the Sorting_Machine_State to get the smallest remaining edge until you find one that doesn't form a cycle with the previously extracted edges.

Besides its use of amortization, the code in Figure 3 is subtle in another respect. It includes a Size operation, which keeps a count of spanning-forest edges without knowing which edges are involved. This means you do not have to compute the minimum spanning forest when Size is first called, which would have unfortunate performance consequences!

To analyze the performance of Figure 3, let n = the number of edges in m. Insert (m,...) takes O(1)time,and Change_To_Extraction_Phase (m) takes $O(n)$ time. Extract (m,...) might take only O(log n) time if the smallest remaining edge is an edge of a minimum spanning forest. If not, it might take much longer, but not more than $O(n \log n)$ time.

On any given graph, both the original implementation of the Find_MST operation and its implementation layered on top of this realization of Spanning_Forest_Machine_Template use $O(n \log n)$ time in the worst case. In summary, there is no difference in performance from the original net-selection problem. However, our recasting design has a potentially significant performance advantage over the conventional design for the bounds-checking problem.

(Incidentally, Figure 3 also uses a Coalesceable_Equivalence_Relation type to solve the cycle-detection problem, and you would want to use it in Find_MST even if you settled for the original design. There are operations to make two integers equivalent and to test if two integers in a Coalesceable_Equivalence_Relation are equivalent, but space prevents us from showing the formal specification for this component here. Suffice to say that an efficient representation of Coalesceable_Equivalence_Relation uses the textbook disjoint-set data structure with path-compression.[2,4])

Conventional reusable component design techniques — even ones based on object-oriented principles —result in components that encapsulate data structures as objects and algorithms as single operations. Separating data structures and algorithms for this purpose is a false dichotomy. Algorithms can and should be encapsulated as objects, just as data structures are. By following machine-oriented design principles, you can achieve more of the functional and performance flexibility potential of systematic component reuse. You also can make your designs consistent and therefore easier for clients to understand.

In principle, there are no limits in applying this approach. For example, you could specify a "record-high" machine that reports each largest item so far; an "eigenvalue" machine that dispenses eigenvalues of a matrix in increasing order;[8] a "compression" or "encryption" machine that works on a series of items.

There are several points to consider when you recast a single large-effect operation as an object. First, try to develop a simple, fully abstract, clearly explainable mathematical model for the collection of items in the machine.[6,7] Then consider if you can settle for a two-phase machine. You probably should have a two-phase version of every machine in the reusable component library even if you can't see the immediate need for it in a particular application. Often, implementations for two-phase machines are easier and/or potentially more efficient than for multiphase or phase-less machines.

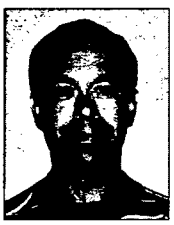
Finally, consider which explanation style you should use to specify the machine's overall behavior by characterizing what it apparently does during the insert, change-to-extraction, and extract operations. What and when your machine does something will depend, in part, on which explanation is most understandable. You might just have to use trial and error before you can judge which is best. But don't worry too much about the initial cost of making these design decisions! If your component is really reusable, the effort you spend on making a good design choice will be amortized over many future uses. ◆

ACKNOWLEDGMENTS

We thank Steve Edwards, Wayne Heym, Joe Hollingsworth, Tim Long, Stu Zweben, and the anonymous *IEEE Software* referees for many helpful suggestions. We also acknowledge the financial support for our research from the National Science Foundation (Weide and Ogden are supported under grants CCR-9111892 and CCR-9311702; Sitaraman is supported under grant CCR-9204461); the Department of Defense's Advanced Research Projects Agency (Weide and Ogden are supported under contract F30602-93-C-0243, monitored by the USAF Materiel Command, Rome Laboratories, ARPA order A714; Sitaraman is supported under ARPA contract DAAH04-94-G-0002, monitored by the US Army Research Office); and the National Aeronautics and Space Administration (Sitaraman is supported under grant 7629/229/0824).

REFERENCES

1. B.W. Weide, W.F. Ogden, and S.H. Zweben, "Reusable Software Components," *Advances in Computers Vol. 33*, M.C. Yovits, ed., Academic Press, New York, 1991, pp. 1-65.
2. T.H. Cormen, C.E. Leiserson, and R.L. Rivest, *Introduction to Algorithms*, MIT Press, Cambridge, Mass., 1990.
3. D.L. Parnas, "On the Criteria to Be Used in Decomposing Systems Into Modules," *Comm. ACM*, Dec. 1972, pp. 1,053-1,058.
4. E. Horowitz and S. Sahni, *Fundamentals of Data Structures*, Computer Science Press, Rockville, Md., 1976.
5. M. Sitaraman, L.R. Welch, and D.E. Harms, "On Specification of Reusable Software Components," *Int'l J. of Software Eng. and Knowledge Eng.* June 1993, pp. 207-219.
6. B.W. Weide et al., "Design and Specification of Iterators Using the Swapping Paradigm," *IEEE Trans. Software Eng.*, Aug. 1994.
7. Special feature on "Component-Based Software Using Resolve," *SIGSoft Software Eng. Notes*, Oct. 1994, to appear.
8. E.R. Davidson, "Monster Matrices: Their Eigenvalues and Eigenvectors," *Computers in Physics*, Sept./Oct. 1993, pp. 519-522.



Bruce W. Weide is an associate professor of computer and information science at Ohio State University, and codirector of the Reusable Software Research Group with Bill Ogden and Stu Zweben. His research interests include all aspects of software component engineering, especially in applying RSRG work to Ada and C++ practice.

Weide received a BS in electrical engineering from the University of Toledo and a PhD in computer science from Carnegie Mellon University. He is a member of the IEEE, ACM, and Computer Professionals for Social Responsibility.

William F. Ogden is an associate professor of computer and information science at Ohio State University, and codirector of the Reusable Software Research Group with Bruce Weide and Stu Zweben. His main research interests are in software reuse, software specification, and program verification.

Ogden received a BS in mathematics from the University of Arkansas and an MS and a PhD in mathematics from Stanford University. He is a member of the IEEE Computer Society and ACM.

Murali Sitaraman is on the faculty of statistics and computer science at West Virginia University. His research interests span all areas of software component construction including design, formal specification, and verification.

Sitaraman received an ME in computer science from the Indian Institute of Technology at Bangalore and a PhD in computer and information science from Ohio State University.

Address questions about this article to Weide at Ohio State University, Computer and Information Science Dept., Columbus, OH 43210; weide@cis.ohio-state.edu.

The Effects of Layering and Encapsulation on Software Development Cost and Quality

Stuart H. Zweben, Stephen H. Edwards, Bruce W. Weide, and Joseph E. Hollingsworth

Abstract— **Software engineers often espouse the importance of using abstraction and encapsulation in developing software components. They advocate the "layering" of new components on top of existing components, using only information about the functionality and interfaces provided by the existing components. This layering approach is in contrast to a "direct implementation" of new components, utilizing unencapsulated access to the representation data structures and code present in the existing components. By increasing the reuse of existing components, the layering approach intuitively should result in reduced development costs, and in increased quality for the new components. However, there is no empirical evidence that indicates whether the layering approach improves developer productivity or component quality.**

We discuss three controlled experiments designed to gather such empirical evidence. The results support the contention that layering significantly reduces the effort required to build new components. Furthermore, the quality of the components, in terms of the number of defects introduced during their development, is at least as good using the layered approach.

Experiments such as these illustrate a number of interesting and important issues in statistical analysis. We discuss these issues because, in our experience, they are not well-known to software engineers.

Index Terms—**Empirical study, encapsulation, software components, abstract data types, software development, software reuse.**

I. Introduction

IT IS well-known that software systems typically exceed their expected development and maintenance costs. While there are many perceived reasons for this, one reason is that the components of these systems tend to be developed nearly from scratch, instead of being predominantly reuses of existing components [15].

The ability to successfully reuse components in new systems depends on the components being properly encapsulated. That way, new components can be "layered" on top of them, taking advantage of the abstract notions already encapsulated and avoiding reimplementation of these abstractions. For many years, programming languages have provided various means of encapsulating data structures and their associated operations, and the means of layering new components using these encapsulated components. Supposedly, there are productivity and quality gains to be had by following this layering technique [20].

However, currently we do not see in software systems the widespread use of encapsulation to layer the system's components. This is true even in popular component libraries. For example, Booch [2] represents a map abstraction as a hash table using chaining for collision resolution. But he codes from scratch the lists that represent chains. He does not reuse his list package. Studies of object-oriented class hierarchies often have found that these "hierarchies" in fact are very flat [4], indicating that most components are not really built on top of existing components.

One possible reason for this lack of layering is that the original components are not well-designed, so that 1) the components needed in the next application are not quite those that are currently available, and 2) the available components cannot easily be converted (by layering) into those that *are* needed. Features and standard use of programming languages, such as Ada's restriction on the mode for parameters to functions and mixed use of private and limited private types, also inhibit one's ability to compose components [11].

Another reason may be that it is not obvious that, even if there were a useful component available, it would be better to reuse it by respecting its encapsulation than it would be to develop the required new functionality by accessing the underlying representation of the component. Performance is most often mentioned as the basis for this belief, although in our experience the performance penalty typically is minimal if the proper abstract functionality is encapsulated in the component. Language features such as code inheritance actually encourage violation of encapsulation while appearing to support layering [12].

We know of no controlled studies showing that the use of layering and encapsulation improves the cost of component development or the quality of the components, based on such measures as time to design/develop or number of defects in the resulting product. While abstract and anecdotal arguments have some place in technology assessment, it is vital that sound empirical support be obtained for the use of emerging software technologies.

Manuscript received April 1994; revised September 1994 and November 1994. Recommended by J. Gannon. This work was supported in part by the National Science Foundation under Grants CCR-9111892 and CCR-9311702, and by ARPA under Contract F30602-93-C-0243, monitored by the U.S. Air Force Material Command, Rome Laboratories, ARPA Order A714.

S. H. Zweben, S. H. Edwards, and B. W. Weide are with Department of Computer and Information Science, The Ohio State University, Columbus, OH 43210 USA (e-mail: zweben@cis.ohio-state.edu); (e-mail: edwards@cis.ohio-state.edu); (e-mail: weide@cis.ohio-state.edu).

J. E. Hollingsworth is with the Department of Computer Science, Indiana University Southeast, New Albany, IN 47150 USA (e-mail: jholly@ius.indiana.edu).

IEEE Log Number 9409037.

Lewis *et al.* [13], did study differences in productivity and quality when subjects were allowed to reuse existing components versus when they were not. They found that subjects who were allowed to reuse performed significantly better than those who were not. However, for their experiments, "reuse" included alteration of existing components' code. Therefore, these experiments really could not show the effect of layering. The Lewis experiments also studied the effects due to the language in which the components were written, using an object-oriented language (C++) or a procedural language (Pascal). While there undoubtedly were some potential differences in encapsulation between these alternatives, the encapsulation did not need to be respected by the subjects in reusing components, since the subjects were allowed to see inside the existing encapsulated units and modify them as they saw fit.

In this paper, we report on three experiments designed to assess the effect, on effort and quality, of designing and developing software components by taking full advantage of existing abstractions, compared with an approach that allows seeing and using information about the implementation of the existing abstractions when designing and developing new components. Each of the experiments is a controlled study [5], and in each study there are interesting aspects to the statistical analysis required to determine if there is a significant difference between the two approaches.

The next section gives some additional background to motivate the experiments, while Section III describes each experiment in detail. Section IV discusses important issues that affect the statistical analyses appropriate for these experiments. Section V discusses the results.

II. Background

Controlled studies are relatively rare in software engineering, and this often is attributed to the prohibitive cost of repeating large systems development tasks so that the effect of using a particular method can be examined relative to a baseline. Most attempts at doing a controlled analysis have been done with small sets of programmers (frequently students) on very small projects. While the applicability of the results of such studies to "real" systems is not obvious, studies of this nature have proven useful in the past. In some cases, they have shown that a particular method may not have the desired effect, or they have helped us to understand better the various factors that may influence the effect of a method [19]. If the tasks performed by subjects in these experiments indeed are a subset of the tasks performed in the development of larger systems, and if the results of these experiments can be replicated, then the experiments can offer useful information about pieces of the complex process of software development.

What kinds of tasks might be typical of large systems development, and therefore worth investigating via controlled experimental studies? Two possibilities are 1) adding functionality to an existing component, and 2) building a component that has *similar* functionality to that of an existing component. In the former case, all that is desired is a new set of operations, but the existing component is of the right kind to provide these operations; the existing component's set of capabilities just needs to be enhanced. In the latter case, a closely related component might be available, but the desired component must provide somewhat different, rather than merely additional, functionality.

TABLE I
COMPARISON OF COMPONENT DEVELOPMENT APPROACHES

Component Devel. Approach	Type of Reuse	Information Used	Understanding Required
Direct Implementation	White box (open, transparent)	Coding details, representation data structures	Purpose/functionality and implementation of existing components
Layered	Black box (closed, opaque)	Functional specification, interface description	Purpose/functionality of existing components

In both types of tasks, one can imagine developing the new components by reusing existing ones in at least two ways. One reuse approach is to "directly implement" the desired functionality using the coding details of existing components. This involves making use of information about the data structures used to represent the objects encapsulated by the existing components. Some people refer to this type of reuse as "white box," "open," or "transparent" reuse. A second reuse approach to developing the desired functionality is to use only the specifications of the functionality of, and the interfaces to, the existing components. We call this the "layering" approach because the new functionality is provided by components built on top of the existing encapsulated units. Other names for this approach might be "black box," "closed," or "opaque" reuse (see Table I).

Both approaches require that the implementor understand the purpose of and functionality provided by the existing components. But the first approach also requires an understanding of the *implementation* of the existing components, while the second approach does not. Intuitively, this suggests that black-box reuse demands of the implementor a lower cognitive load, and hence should reduce the *effort* required, at least for the initial design and coding of the new components.

The expected effect on *quality* of layering versus direct implementation is somewhat less clear. In this paper, we use the term "quality" in the sense of "correctness," though we know that quality has other dimensions, too. Since the need to work with, and possibly misuse, the existing implementation is eliminated with black box strategies, one might expect better quality (i.e., fewer defects) from components developed using layering. On the other hand, since with layering one has available only the operations provided by the existing components and not the underlying representation data structures, the algorithms used in the layering approach might differ from those used in the direct implementation approach. This use of different algorithmic approaches might give rise to different distributions of defects. Even if the same algorithmic approaches are used in the two methods, most of the defects might occur in the attempt to synthesize the algorithm from the existing operations, rather than in the manipulation of representations of existing data structures.

Both of the tasks described above, that of adding functionality and that of modifying functionality, and both reuse

approaches, were studied in our experiments. The experiments are described in detail in the following section.

III. Overview of the Experiments

Our experiments were conducted as part of a course on the subject of "Software Components in Ada." This course was offered in two separate quarters, once in the summer of 1991 and again in the fall of 1992. In each case, the students in the course were graduate and upper-division undergraduate students majoring in computer science. Several of the students were full-time employees in the computing field.

The lectures heavily emphasized the trade-offs evident with principles of encapsulation, abstraction and layering, and presented a detailed engineering discipline for designing, formally specifying, and correctly and efficiently implementing Ada generic packages exporting abstract data types. Several programming assignments illustrated main points from the lectures, and the experiments were conducted in conjunction with some of these assignments.

Our main independent variable for the experiments was the reuse approach used by the subjects, and the dependent variables of interest were effort and component quality. Therefore, the primary null hypothesis to be tested is that the reuse approach used has no significant effect on the development effort or the quality of the resulting component.

To assess effort, we had the subjects keep careful records of the time spent in the initial designing/coding, testing, and debugging/recoding phases of the task. These times were recorded individually for each operation exported by the component under investigation. Both the initial designing/coding time and the total time to complete the task (including the testing and debugging/recoding time) were of interest to this study.

To assess quality, for each operation of the component the subjects recorded the number of defects they needed to fix. Only those defects that caused run-time failures during testing were included.

We emphasized to the students the importance of being internally consistent in keeping and reporting the data, and stressed that grades in the course would have nothing to do with the reported numbers. We audited the information provided by the students through post-assignment interviews, and the programs submitted by the students were tested by the instructor to ensure (as well as possible) that no lingering defects remained. We were somewhat skeptical of the data on a per-operation basis, but were confident that the aggregate information provided by the students was accurate.

Several factors other than reuse approach might influence the results on the dependent variables. One of these, the nature of the activity (i.e., enhancement of functionality versus modification of functionality) was mentioned in the previous section. Other important factors include the component involved in the reuse activity, the subjects' familiarity with the component specification and representation, and the abilities of the subjects. In planning the experiments, we attempted to deal with each of these issues.

The first experiment was an "enhancement of functionality" exercise using a simple, well-understood unbounded queue component. In addition to the "universal" package operations of `Initialize`, `Finalize`, and `Swap` [21], [8], [11], the basic Ada queue package provided the standard `Enqueue`, `Dequeue`, and `Is_Empty` operations (specifications for an Ada queue package can be found in [11]). The enhancement task was to add operations to `Copy` a queue, `Clear` (i.e., empty) a queue, `Append` one queue to another, and `Reverse` a queue. The representation structure used for the queue was a standard linked data structure with pointers to the front and rear. Eighteen subjects participated in this experiment. Since the functionality and representation structure used in this package were so familiar to the subjects—both from classical data structures and a previous lab exercise—we felt that it would be difficult to obtain significant differences due to the approach used. Preliminary results from this experiment were reported in [10].

The second experiment was a replication of the first experiment using a more complex component that encapsulated a "partial map." This component allows a client to create an associative mapping which is a partial function from an arbitrary domain type to an arbitrary range type; it is useful in table processing applications. In addition to `Initialize`, `Finalize`, and `Swap`, the basic package provided the following operations (specifications for an Ada partial map package can be found in [18]).

- `Make_Defined` Assign a given range value to a given domain element (add an association to the map).
- `Make_Undefined` Undefine a given, presently defined domain element (i.e., remove a particular association from the map).
- `Make_Any_One_Undefined` Undefine some presently defined domain element, chosen arbitrarily and returned by the implementation (remove an arbitrary association from the map).
- `Test_If_Defined` Test if a given domain element is defined.
- `Test_If_Any_One_Defined` Test if there are *any* defined elements in the map.

The enhancement task for this experiment was to add operations that would `Display` (print) the map, `Clear` it (making every domain element undefined, according to the map), `Combine` two partial maps (assuming there were no inconsistently defined mappings in these two maps), and `Remap` all domain elements that map to one range value so they instead map to another range value. The representation chosen for the partial map was a hash table. The subjects were taught about this package in a previous lab assignment, where they were asked to implement the package using the hash table representation. Nevertheless, in comparison with the unbounded queue package, both the specification of the partial map package (i.e., the description of the functionality of the operations) and the representation of the data type were more complicated. Furthermore, we felt that the experience with the representation structure gained in the earlier lab could only *improve* the subjects' abilities to do the direct implementation

version of the enhancement, relative to their ability to do the layered version. Again, we expected that it would be difficult to obtain significant differences between the reuse approaches. Ten subjects participated in this experiment.

The third experiment was a "modification of functionality" experiment using the partial map component. Given the basic partial map package used in the second experiment, the task was to create a component that encapsulated the concept of an *almost constant map*, which maps all but a finite number of domain elements to a "default" range value. Other than `Initialize`, `Finalize`, and `Swap`, the operations required by this package are as follows.

- `Reset` Reset the map to a constant function in which all domain elements map to a specified default value.
- `Get_Default` Return the default value, to which most of the elements are mapped.
- `Swap_Range_Value` Modify the range value associated with a given domain element.
- `Remove_Any_Anomaly` Make an arbitrary domain element, presently not mapping to the default value, instead map to the default value.
- `Test_If_Anomaly` Test if a given domain value maps to the default value.
- `Test_If_Constant` Test if all domain values map to the default value.

In this experiment, the subjects would be creating a rather unfamiliar component using what, by now, would be a somewhat familiar component. The same ten subjects who participated in the second experiment also participated in this third experiment.

To mitigate the differential effects of subjects on task completion (see, e.g., [3]), in each experiment each subject performed the task using *both* the layering and direct implementation approaches. The order in which the two approaches were followed was assigned randomly, subject to a counterbalancing with respect to the experience level of the subjects (i.e., we didn't want more experienced subjects doing the tasks in one order while the less experienced subjects did the tasks in the other order). Due to the uneven distribution of subject experiences and the relatively small number of subjects available for study, we chose not to investigate experience level as a separate factor in the experimental design.

To summarize, each experiment was designed to assess the effect of the reuse approach (layered or direct implementation) on each of three dependent variables (initial design and coding time, total time, and number of defects). Each subject in each experiment was assigned to one of two sequences in which the two reuse approaches were employed (layered first or direct implementation first). Thus, as illustrated in Table II, each of the experiments was a two-period crossover design [14], [16].

TABLE II
FACTORS IN THE EXPERIMENTAL DESIGN

	Sequence	
Treatment	Layered First	Direct First
Layered	9 subj. in Exp. 1	9 subj. in Exp. 1
Direct	5 in Exp. 2 and 3	5 in Exp. 2 and 3

IV. STATISTICAL ANALYSIS ISSUES

The analysis of variance (ANOVA) model for these experiments allows us to decide if there is a significantly different effect, on each dependent variable, of the layering approach versus the direct implementation approach (in statistical terminology, this is the "treatment effect" in the experiment). The null hypothesis is that there is no such effect. In addition, the model allows us to decide if there is a significant difference in the two possible sequences (orders) in which these two approaches were employed. Finally, it allows us to assess if the treatment and the sequence interact; that is, we can test if there is a different effect of the approach depending on the order in which the two approaches were employed. Since each subject did the same task twice (once for each of the two approaches), but did so in only one of the two possible sequences, in statistical terminology the subjects are nested within sequences. Thus, this model is sometimes called a "nested factorial" design [9]. In the model, the variance associated with subjects (nested in sequences) is used to test for the sequence effect, while the variance of the subject by treatment interaction is used to test for the treatment effect and the treatment by sequence interaction. Some texts call this latter variance the "time error term" [14].

However, the normal nested factorial analysis of variance may not be appropriate for these experiments, for two reasons.

1) An important concern is the potential that a subject's having done a task once will affect performance on the second attempt at the task (even though a different approach is being used). This possibility of a "carryover effect," as it is known in statistics, requires that care be taken when deciding if there really is an effect due to the approach used.

A common and accepted way of dealing with this issue is first to test if there is a significant sequence effect and/or a significant interaction effect. If there is neither, then the normal nested factorial analysis of variance is used to test for the effect of the treatment (i.e., the approach). If either the sequence effect or the interaction effect is significant, however, the treatment effect is tested using only the data for the first period in the sequence. That is, for those subjects who did the layering approach first, only their data for the layering approach is used; for those who did the direct implementation first, only their data for the direct implementation approach is used. The error estimate used for this test is a function of the subject error and time error terms [14].

2) The defect data typically will comprise small integer values, whose distribution does not satisfy the normality assumptions required of a standard analysis of variance. Defect data may be modeled more accurately by a Poisson-like distribution, and there are common statistical procedures to do Poisson (regression) analysis, allowing tests for the significance of the sequence, treatment, and sequence by treatment interaction effects. However, a true Poisson distribution would imply that the mean and variance are equal. The well-known vast differences in subjects makes this assumption unlikely to be met in our experiments. When the true variance is higher than that assumed by the Poisson model, a phenomenon called

"overdispersion" is present. The statistical analysis therefore must deal not only with nonnormality, but also with overdispersion. Without correcting for overdispersion, for example, the test for significance of the treatment (approach) may falsely indicate significance. Fortunately, there are statistical procedures to deal with this [1]. These procedures compute test statistics to see if a true Poisson regression is appropriate, or if a correction for overdispersion must be applied.

Some authors [5], [7] suggest, when analyzing data for which the assumptions of normality are questionable[1], that nonparametric tests may be more appropriate. However, it also is known that parametric tests on means and in experiments where cell sizes are equal (characteristics of our experiments), are fairly robust against deviations from normality [17]. Moreover, though it is easy to perform standard nonparametric tests such as the Mann-Whitney test [5], [6] to see if the primary effect due to reuse approach is present, these tests lose other information present in the experimental design and in the data (e.g., order, nesting of subjects within order, possible interactions between order and treatment, and actual values of the time and number of defects instead of their ranks). The Mann-Whitney test also is not very powerful for situations where there are many tied scores (as we have with the defect data), though it is possible to adjust for ties. Parametric models therefore allow one to get more information from the data, and are preferred if the models' assumptions are reasonably met.

In the following section, we report the detailed analyses of the parametric tests only. We did perform Mann-Whitney tests on the treatment effect for each experiment. The results of these nonparametric tests were exactly the same as their parametric counterparts, at the same significance level.

V. Results

While there are enough data points to apply the statistical methods used in this paper, the small sample sizes used in our experiments adversely affect the power of the statistical tests. This means that, if indeed there are real differences between layering and direct implementation, our tests might conclude otherwise. One way to improve the power of the tests is to raise the alpha level (the probability of concluding that there is a significant treatment effect when in fact there is none). In each of the analyses that follow, the significance tests are done using the standard alpha level of .05, though a case might be made that a higher alpha level (e.g., .10) would be reasonable.

In the first experiment, the subjects already were very familiar with the component in question (the queue). They had seen, and likely used several times, the standard linked representation structure used in this experiment. Hence, the simplicity and familiarity of the component itself might mask true effects due to the treatment. Prior to the second and third experiments, the partial map component used in these experiments had been studied by the subjects in an exercise in which they did an implementation using hashing. A hashing implementation was, in fact, used as the representation of the partial map component provided in experiments two and three (though the actual code for the implementation used in the experiments likely differed somewhat from that developed by any of the subjects in their exercise).

By using the standard .05 alpha level we felt that, if our experiments were biased at all, they were biased in favor of *not* getting a significant treatment effect when in fact one was present. Therefore, our judgment was that, if the experiment showed a significant treatment effect, it was not likely to be spurious.

[1] Our time data also could fall into this category. There is some positive skewness in the data; actual tests for normality are highly volatile on small samples such as we have in our experiments.

TABLE III
EXPERIMENT 1—INITIAL DESIGN/CODING TIME DATA SUMMARY

Treatment	Sequence				Overall	
	Layered First		Direct First			
	Mean	SD	Mean	SD	Mean	SD
Layered	99.1	41.8	57.2	21.3	78.2	38.7
Direct	94.0	44.3	180.4	65.4	137.2	70.1
Overall	96.6	41.9	118.8	79.0	107.7	63.4

TABLE IV
EXPERIMENT 1—INITIAL DESIGN/CODING TIME ANALYSIS

Source	df	Mean Sq.	f
Sequence	1	4466.7	1.84
Subject (within Sequence)	16	2434.7	
Sequence × Treatment	1	37056.3	20.73*
Subject × Treatment (within Sequence)	16	1787.5	
Treatment (first period only)	1	6609.7	14.09*
Error	16	469.1	

A. Experiment 1—Queue Enhancement

The analysis of variance for this experiment, using initial design and coding time, revealed no significant effect for sequence (mean for layering first = 96.6 min., mean for direct first = 118.8 min., $f = 1.84$, critical $F_{1,16,.95} = 4.49$), but did indicate a significant sequence by treatment interaction ($f = 20.73$, $F_{1,16,.95} = 4.49$). The means and standard deviations for each of the four cells are shown in Table III, while the relevant components of the ANOVA table are shown in Table IV. In the ANOVA tables, statistically significant f values (at the .05 level) are indicated by an asterisk.

Following the approach outlined in the previous section, the treatment effect was tested using only the first period data, with the result that the layering approach required significantly less effort ($f = 14.09$, $F_{1,16,.95} = 4.49$).

When total time was used as the dependent variable, a situation similar to that for initial design and coding time was observed. There was no significant effect for sequence (mean for layering first = 163.4 min., mean for direct first = 182.6 min., $f = 0.30$, $F_{1,16,.95} = 4.49$), there was a significant sequence by treatment interaction ($f = 7.16$, $F_{1,16,.95} = 4.49$), and the first period data showed that the layering approach was significantly faster ($f = 7.81$, $F_{1,16,.95} = 4.49$). Tables V and VI contain the relevant data and ANOVA, respectively, for total time.

TABLE V
EXPERIMENT 1—TOTAL TIME DATA SUMMARY

Treatment	Sequence				Overall	
	Layered First		Direct First			
	Mean	SD	Mean	SD	Mean	SD
Layered	145.1	81.4	102.8	47.6	123.9	68.2
Direct	181.8	111.0	262.3	102.4	222.1	111.5
Overall	163.4	96.3	182.6	112.8	173.0	103.8

TABLE VI
EXPERIMENT 1—TOTAL TIME ANALYSIS

Source	df	Mean Sq.	f
Sequence	1	3287.1	0.30
Subject (within Sequence)	16	11087.7	
Sequence × Treatment	1	33878.8	7.16*
Subject × Treatment (within Sequence)	16	4748.5	
Treatment (first period only)	1	13735.8	7.81*
Error	16	1759.6	

TABLE VII
EXPERIMENT 1—DEFECT DATA SUMMARY

Treatment	Sequence				Overall	
	Layered First		Direct First			
	Mean	SD	Mean	SD	Mean	SD
Layered	0.89	0.93	1.33	1.41	1.11	1.18
Direct	2.67	3.57	4.22	3.99	3.44	3.76
Overall	1.78	2.69	2.78	3.26	2.28	2.99

TABLE VIII
EXPERIMENT 1—DEFECT ANALYSIS (POISSON REGRESSION)

Source	df	Mean Sq.	f
Sequence	1	1.38	0.48
Treatment	1	7.84	2.72
Sequence × Treatment	1	<0.01	<0.01
Error	32	2.88	

The Poisson analysis of the defect data revealed neither a significant sequence effect (mean for layering first = 1.78, mean for direct first = 2.78, $f = 0.48$, $F_{1,32,.95} = 4.16$) nor a significant sequence by treatment interaction ($f < 0.01$, $F_{1,32,.95} = 4.16$). Tables VII and VIII contain the relevant statistics. There was no significant treatment effect after correcting for overdispersion ($f = 2.72$, $F_{1,32,.95} = 4.16$).

B. Experiment 2—Partial Map Enhancement

The analysis of initial design and coding time for the partial map enhancement was similar to that for the queue enhancement. There was no significant sequence effect, but there was a significant sequence by treatment interaction (Tables IX and X). The test for the treatment effect, using only the first period data, revealed that the layering approach required significantly less effort ($f = 18.87$, $F_{1,8,.95} = 5.32$).

The analysis of total time gave results similar to that of initial design and coding time. Tables XI and XII show the data and analysis; the treatment test using only the first period data was significant in favor of layering ($f = 14.35$, $F_{1,8,.95} = 5.32$). Again, this replicated the results of Experiment 1.

The Poisson analysis of the defect data showed no significant effect either for sequence or sequence by treatment interaction. The treatment test, after correcting for overdispersion, showed a significant difference in favor of layering (i.e., the layering approach gave rise to significantly fewer defects). This result differed from that in Experiment 1, where no significant effect was observed (see Tables XIII and XIV for details).

TABLE IX
EXPERIMENT 2—INITIAL DESIGN/CODING TIME DATA SUMMARY

Treatment	Sequence				Overall	
	Layered First		Direct First			
	Mean	SD	Mean	SD	Mean	SD
Layered	69.8	16.6	119.0	103.8	94.4	74.7
Direct	127.6	29.6	273.2	131.1	200.4	118.0
Overall	98.7	37.9	196.1	138.0	147.4	110.4

TABLE X
EXPERIMENT 2—INITIAL DESIGN/CODING TIME ANALYSIS

Source	df	Mean Sq.	f
Sequence	1	47433.8	3.63
Subject (within Sequence)	8	13082.6	
Sequence × Treatment	1	11616.2	7.86*
Subject × Treatment (within Sequence)	8	1477.5	
Treatment (first period only)	1	41371.6	18.87*
Error	8	2912.0	

TABLE XI
EXPERIMENT 2—TOTAL TIME DATA SUMMARY

Treatment	Sequence				Overall	
	Layered First		Direct First			
	Mean	SD	Mean	SD	Mean	SD
Layered	113.0	62.9	160.6	147.3	136.8	109.7
Direct	191.0	40.5	458.2	236.1	324.6	212.9
Overall	152.0	64.6	309.4	242.9	230.7	190.9

TABLE XII
EXPERIMENT 2—TOTAL TIME ANALYSIS

Source	df	Mean Sq.	f
Sequence	1	123873.8	3.37
Subject (within Sequence)	8	36725.2	
Sequence × Treatment	1	60280.2	12.59*
Subject × Treatment (within Sequence)	8	4786.8	
Treatment (first period only)	1	119163.0	14.35*
Error	8	8302.4	

TABLE XIII
EXPERIMENT 2—DEFECT DATA SUMMARY

Treatment	Sequence				Overall	
	Layered First		Direct First			
	Mean	SD	Mean	SD	Mean	SD
Layered	0.80	1.79	0.80	0.84	0.80	1.32
Direct	2.60	1.67	6.60	2.61	4.60	2.95
Overall	1.70	1.89	3.70	3.56	2.70	2.96

TABLE XIV
EXPERIMENT 2—DEFECT ANALYSIS (POISSON REGRESSION)

Source	df	Mean Sq.	f
Sequence	1	4.24	2.37
Treatment	1	16.52	9.23*
Sequence × Treatment	1	0.79	0.44
Error	16	1.79	

TABLE XV
EXPERIMENT 3—INITIAL DESIGN/CODING TIME DATA SUMMARY

Treatment	Sequence				Overall	
	Layered First		Direct First			
	Mean	SD	Mean	SD	Mean	SD
Layered	65.4	28.8	50.6	33.5	58.0	30.5
Direct	112.4	65.1	133.8	52.6	123.1	56.9
Overall	88.9	53.5	92.2	60.4	90.6	55.6

TABLE XVI
EXPERIMENT 3—INITIAL DESIGN/CODING TIME ANALYSIS

Source	df	Mean Sq.	f
Sequence	1	54.5	0.02
Subject (within Sequence)	8	3609.0	
Treatment	1	21190.1	24.42*
Sequence × Treatment	1	1638.1	1.89
Subject × Treatment (within Sequence)	8	867.8	

TABLE XVII
EXPERIMENT 3—TOTAL TIME DATA SUMMARY

Treatment	Sequence				Overall	
	Layered First		Direct First			
	Mean	SD	Mean	SD	Mean	SD
Layered	157.8	99.4	112.0	83.6	134.9	89.9
Direct	193.2	112.9	311.8	327.4	252.5	239.2
Overall	175.5	102.0	211.9	248.7	193.7	185.9

TABLE XVIII
EXPERIMENT 3—TOTAL TIME ANALYSIS

Source	df	Mean Sq.	f
Sequence	1	6624.8	0.13
Subject (within Sequence)	8	49544.4	
Treatment	1	69148.8	3.67
Sequence × Treatment	1	33784.2	1.79
Subject × Treatment (within Sequence)	8	18859.9	

C. Experiment 3—Partial Map Modification

In the analysis of initial design and coding time for the partial map modification experiment, there was no significant effect for sequence nor for the sequence by treatment interaction. The analysis of the treatment effect, using all of the data, revealed a significant treatment effect in favor of layering (Tables XV and XVI).

The analysis of total time also revealed no significant sequence effect, nor a significant sequence by treatment interaction. The treatment effect was not significant (Tables XVII and XVIII).

TABLE XIX
EXPERIMENT 3—DEFECT DATA SUMMARY

Treatment	Sequence				Overall	
	Layered First		Direct First			
	Mean	SD	Mean	SD	Mean	SD
Layered	3.00	2.45	1.00	1.22	2.00	2.11
Direct	5.00	4.85	2.80	2.49	3.90	3.81
Overall	4.00	3.77	1.90	2.08	2.95	3.15

TABLE XX
EXPERIMENT 3—DEFECT ANALYSIS (POISSON REGRESSION)

Source	df	Mean Sq.	f
Sequence	1	2.57	0.84
Treatment	1	2.09	0.70
Sequence × Treatment	1	0.25	0.09
Error	16	2.98	

The Poisson analysis of the defect data revealed no significant sequence effect nor a significant sequence by treatment interaction. After correcting for overdispersion, the test on treatment was not significant (Tables XIX and XX).

D. Discussion

For this set of experiments, we found that the use of layering consistently resulted in significantly faster time to complete the initial design and coding of the new component. This result held as the component changed from one with which the subjects were quite familiar to one with which the subjects were less familiar (and which was more complex as measured by the number of operations encapsulated in it and total lines of code). The result also held as the task changed from a component enhancement to a component modification.

Total time, including that for testing, debugging and recoding, also was significantly better using layering when the component was an enhancement. This result held for both the simple and familiar queue, and the more complicated and less familiar partial map. The modification task provided no such significant effect, though the mean total time for layering was much less than that for direct implementation. One possible explanation for this is that some defects made in the modification task tended to be nastier than those made in the enhancement task. It turned out that the average number of defects per subject was slightly higher for the enhancement task than for the modification task. If some of the defects for the modification task were, indeed, trickier, the time to debug and repair these defects would occupy a greater fraction of total time. Apparently, the *nature* of the defects was such that this debugging and repair time was not a function of the treatment, so the gain for layering in initial design and coding time is ameliorated when the debugging and repair time is added. Note that this could mean that trivial defects were just as trivial and nasty defects were just as nasty, whether layering or direct implementation was used.

No consistent effects in favor of layering were found for the defect data, though the mean number of defects was less

for layering in each experiment. Here the small scale of the experiment may influence the results. Almost every subject had fewer than five defects. Some of these defects likely were trivial and these relatively trivial defects might be just as likely to be made when using direct implementation as they are when using layering. If the fraction of relatively trivial defects was high for many of the subjects, it then will be difficult to obtain statistical significance. With the small number of defects observed in these data, it is somewhat remarkable that we obtained *any* significant effects for this dependent variable.

A final item worth noting is that, had the analysis of the defect data for experiment 1 *not* corrected for overdispersion, it would falsely have concluded that there *was* a significant effect favoring layering. This illustrates how using the wrong statistical analysis in software engineering experiments such as these can mistakenly support the use of a particular technology even when the data do not really indicate such support.

VI. Conclusion

The results of our experiments support the contention that, by using only a description of the functionality of and interfaces to existing components, new components can be developed with less effort than that required if the source code and representation data structures of the existing components are also used. In addition, it appears from our experiments that there certainly is no loss in the quality of the development process, at least in terms of the number of defects made during development, when the layering approach is used.

The empirical studies described in this paper illustrate interesting issues in the statistical analysis procedure, issues which, based on the authors' experiences, are not well-known to software engineering researchers. It is important that the proper analysis is used, lest the wrong conclusions be reached regarding the benefits of a particular method and/or best use is not made of the information contained in the experimental design and the data collected.

The subjects used in our experiments, while mature students many of whom had full-time jobs involving software development, might not be representative of the typical programmer. Generally, they had only a couple of years' experience in commercial software development. Subjects with different backgrounds might perform differently on our experimental tasks; this is a potential avenue for future research.

It also might be interesting to compare the actual amount of code written by the subjects when using layering with that written when using direct implementation, to see if layering required less "work." If so, then the amount of work required (measured by required changes to the code) would be an alternative explanation of our results for the time and defect data. We did not collect this "code change" data. Of course, from an abstract point of view, the amount of change required when using layering was identical to that required when using direct implementation, since the functionality required was the same in each case. Moreover, we believe that a software engineer, when faced with the choice of using layering or direct implementation, would find it difficult to estimate in advance which approach would require less work, even when (as in our experiments) the engineer is quite familiar with the representations used in the direct implementation.

We are planning further experiments to more carefully analyze the defects made in the development of components such as those studied herein. It is important not only to characterize the kinds of defects observed, but also to provide if possible some cognitive explanation of these observations. We also are planning other studies to try to replicate the results reported herein. Studies such as these serve to provide a more sound and scientific basis for using (or not using) various software engineering methods.

Acknowledgment

The authors thank R. Leighty of Ohio State University's Department of Statistics for his assistance with the statistical analyses, and the referees and editor for their helpful comments.

References

[1] M. Aitkin *et al.*, *Statistical Modeling in GLIM*. New York: Oxford, 1989.
[2] G. Booch, *Software Components in Ada*. Menlo Park, CA: Benjamin Cummings, 1987.
[3] R. Brooks, "Studying programmer behavior experimentally: The problems of proper methodology," *Commun. ACM*, vol. 23, no. 4, pp. 207–213, Apr. 1980.
[4] S. Chidamber and C. Kemerer,"A metrics suite for object-oriented design," *IEEE Trans. Software Eng.*, vol. 20, pp. 476–493, June 1994.
[5] S. Conte, V. Shen, and H. Dunsmore, *Software Engineering Metrics and Models*. Menlo Park, CA: Benjamin Cummings, 1986.
[6] W. Daniel, *Applied Nonparametric Statistics*, 2nd ed. Boston, MA: PWS-Kent, 1990.
[7] N. Fenton, *Software Metrics: A Rigorous Approach*. London, U.K.: Chapman and Hall, 1991.
[8] D. Harms and B. Weide, "Copying and swapping: Influences on the design of reusable software components," *IEEE Trans. Software Eng.*, vol. 17, pp. 424–435, May 1991.
[9] C. Hicks, *Fundamental Concepts in the Design of Experiments*, 2nd ed. New York: Holt, Rinehart and Winston, 1973.
[10] J. Hollingsworth, B. Weide, and S. Zweben, "Confessions of some used-program clients," in *Proc. Fourth Annu. Workshop Software Reuse* (Herndon, VA), Nov. 1991.
[11] J. Hollingsworth, "Software component design-for-reuse: A language-independent discipline applied to Ada," Ph.D. dissertation, Dept. Comput., Info. Sci., Ohio State University, Columbus, OH, Aug. 1992.
[12] W. R. LaLonde, "Designing families of data types using exemplars," *ACM Trans. Programming Languages, Syst.*, vol. 11, no. 2, pp. 212–248, 1989.
[13] J. A. Lewis *et al.*, "An empirical study of the object-oriented paradigm and software reuse," in *Proc. 1991 OOPSLA Conf.*, pp. 184–196.
[14] G. Milliken and D. Johnson, *Analysis of Messy Data, Vol. 1: Designed Experiments*. Princeton, NJ: Van Nostrand Reinhold, 1984.
[15] R. Pressman, *Software Engineering: A Practitioner's Approach*, 3rd ed. New York: McGraw-Hill, 1992.
[16] D. Ratkowski, M. Evans, and J. R. Alldredge, *Cross-Over Experiments*. New York: Marcel Dekker, 1993.
[17] H. Scheffe, *The Analysis of Variance*. New York: Wiley, 1959.
[18] M. Sitaraman and B. Weide, Eds., "Special feature: Component-based software using RESOLVE," *ACM SIGSOFT Software Eng. Notes*, vol. 19, no. 4, pp. 21–67, Oct. 1994.
[19] E. Soloway and S. Iyengar, Eds., *Empirical Studies of Programmers*. Norwood, NJ: Ablex, 1986.
[20] I. Sommerville, *Software Engineering*, 4th ed. Reading, MA: Addison-Wesley, 1992.

[21] B. Weide, W. Ogden, and S. Zweben, "Reusable software components," in *Advances in Computers*, vol. 33, M. C. Yovits, Ed. New York: Academic, 1991, pp. 1–65.

Stuart H. Zweben received the Ph.D. degree in computer science from Purdue University, West Lafayette, IN.

He is Professor and Chair of the Department of Computer and Information Science at The Ohio State University, Columbus. His research interests are in the areas of software quality evaluation and software reuse, and he codirects the Reusable Software Research Group at Ohio State.

Dr. Zweben is current President of the ACM, is former President of the Computing Sciences Accreditation Board, and is a member of the editorial board of the IEEE TRANSACTIONS ON SOFTWARE ENGINEERING. He also is a member of the IEEE Computer Society, ACM, Upsilon Pi Epsilon, and the American Association of University Professors.

Stephen H. Edwards received the B.S.E.E. degree from the California Institute of Technology, Pasadena, and the M.S. degree in computer and information science from The Ohio State University, Columbus, where he is working toward the Ph.D. degree in computer and information science.

Prior to attending Ohio State, he was a Member of the Research Staff at the Institute for Defense Analyses. His research interests are in software engineering and reuse, formal models of software structure, programming languages, and information retrieval technology.

Mr. Edwards is a member of the IEEE Computer Society and the ACM.

Bruce W. Weide received the B.S.E.E. degree from the University of Toledo, Toledo, OH, and the Ph.D. degree in computer science from Carnegie-Mellon University, Pittsburgh, PA.

He is Associate Professor of Computer and Information Science at The Ohio State University, Columbus, and Codirector of the Reusable Software Research Group with B. Ogden and S. Zweben. His research interests include all aspects of software component engineering, especially in applying RSRG work to Ada and C++ practice.

Dr. Weide is a member of the ACM and CPSR.

Joseph E. Hollingsworth received the B.S.C.S. degree from Indiana University, the M.S.C.S. from Purdue University, West Lafayette, IN, and the Ph.D. degree in computer and information science from The Ohio State University, Columbus.

He is an Assistant Professor of Computer Science at Indiana University Southeast, New Albany. His research interests include software component design disciplines for C++ and Ada and the application of those disciplines to real-world computing problems.

Dr. Hollingsworth is a member of the IEEE Computer Society and the ACM.

Proceedings

Fourth International Conference on Software Reuse

April 23-26, 1996 **Orlando, Florida, USA**

Edited by

Murali Sitaraman

Sponsored by

IEEE Computer Society Technical Council on Software Engineering

In cooperation with

Association for Computing Machinery

With support from

Andersen Consulting
Buzzeo Inc.
Digital Equipment Corp.
Intecs Sistemi S.p.A.
Loral Federal Systems
Lucent Technologies
Microsoft
Reuse, Inc.
SAIC
Siemens
SIGS

IEEE Computer Society Press
Los Alamitos, California

Washington • Brussels • Tokyo

Characterizing Observability and Controllability of Software Components

Bruce W. Weide, Stephen H. Edwards, Wayne D. Heym, Timothy J. Long, William F. Ogden
Department of Computer and Information Science
The Ohio State University
Columbus, OH 43210
{weide,edwards,heym,long,ogden}@cis.ohio-state.edu

Abstract

Two important objectives when designing a specification for a reusable software component are understandability and utility. For a typical component defining a new abstract data type, a significant common factor affecting both of these objectives is the choice of a mathematical model of the (state space of the) ADT, which is used to explain the behavior of the ADT's operations to potential clients. There are subtle connections between the expressiveness of this mathematical model and the functions computable using the operations provided with the ADT, giving rise to interesting issues involving the two complementary system-theoretic principles of "observability" and "controllability". This paper discusses problems associated with formalizing intuitively-stated observability and controllability principles in accordance with these tests. Although the example we use for illustration is simple, the analysis has implications for the design of reusable software components of every scale and conceptual complexity.

1. Introduction

Specifying the behavior of a software component — especially one that is meant to be reused — is a challenging task. Some important "quality" objectives of design in this area include avoiding implementation bias [10] and achieving understandability for potential component clients [16]. How can the specifier's design space be limited so high quality reusable component designs are allowed while low quality ones are ruled out? And how can proposed design principles be made effectively checkable and not merely slogans?

Surely no general guidelines can succeed completely, but experience shows that some do constrain the design space in the right ways. In prior work we surveyed several specification principles that were intuitively described in the literature and proposed practical tests for compliance [18]. In this paper we report on some interesting problems associated with two of these principles, observability and controllability, which deal with the relationship between the expressiveness of the mathematics used in a specification and the computational power of the specified component. Informally, they (together) provide a test for "minimality" of the specified state space of an ADT.

Our contributions here are:

- We show why it is important to make careful and unambiguous definitions of these principles, because superficially reasonable interpretations of the informal definitions can easily lead to compliance tests that admit poor designs.
- We illustrate unexpected difficulties in making careful and unambiguous definitions.
- We lay out a road map of possible ways to formalize observability and controllability. At each fork in the road (marked in the text with y) this paper takes a particular branch in concert with folklore about specification design, leading toward and beyond fairly specific principles proposed in the literature [18]. This gives a depth-first view of the landscape of Figure 1. A more comprehensive future paper will discuss the paths we do not follow here.

1.1. The Principle of Observability

One of the most important design decisions facing a reusable component specifier is the selection of an appropriate mathematical model (also called "conceptual model" or "abstract model" or "mental model" [14]) for the state space of values for variables (or "objects") of a new abstract data type (ADT) [3, 8, 17, 18, 20]. This model is used to explain the abstract behavior of a component's operations, so the choice of model directly influences the understandability of the concept and the ease of reasoning about its implementations and clients that are layered on top of it [4, 16]. Typically, the specification designer must consider a variety of candidate mathematical models before identifying the

1085-9098/96 $5.00 © 1996 IEEE

"best" one(s). There are many options because both standard and newly-conceived mathematical models — and compositions and combinations thereof — are candidates.

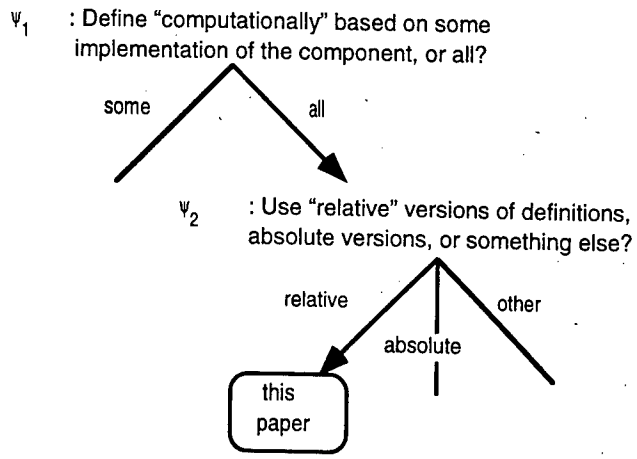


Figure 1 — Major Decision Points in Formalizing Observability and Controllability

An intuitively pleasing ideal that limits the design space in this dimension is the *principle of observability:*

O_0 A specification S defining the program type ADT is observable iff every two unequal values in ADT's state space are "computationally distinguishable" using some combination of operations of S.

An appropriate way to view observability is in terms of the connection between the structure of the state space imposed on it by its mathematical operators and predicates, and the computational structure imposed on it by the specified programming operations. Observability dictates that the model should define a state space which makes distinctions that are just sufficient to specify the intended behavior of the operations — and no more; i.e., the model does not distinguish values that are indistinguishable by the programming operations. One predicate that is available in nearly every useful mathematical state space is equality. Basing observability on equality makes the principle generally applicable, although it is possible to refine it to other predicates particular to individual mathematical theories.

Some designers (e.g., one of the referees of this paper) argue that observability is not an appropriate objective in the first place. For example, consider a simple statistical calculator that provides operations to enter a number and to compute the mean and variance of all numbers entered so far. An intuitively "natural" state space seems to be a multiset of all values entered. But a specification with only the above operations is not observable if based on this state space because many different multisets of numbers can have the same mean and variance. A state space leading to an observable specification for this simple calculator is the number of numbers entered so far, their sum, and the sum of their squares. However, one might argue against this minimal state space on the grounds that it does not support adding a new operation, say, to return the median of the numbers entered so far.

This argument might seem persuasive for traditional software design where one must add such an additional operation using cut-and-paste of source code. But it is inapplicable to a "black-box" component reuse technology such as we are discussing [18]. The simple statistical calculator with only mean and variance operations cannot be used to compute the median without breaking under the covers of the calculator to change its internal representation. This fact demonstrates that the proposed simple calculator is simply not an appropriate reusable component if the requirement is to find the median of a set of numbers. This client should choose a more powerful calculator component.

A prime motivation for demanding observability as a property of truly reusable components is a psychological one. In trying to understand a specification, a client naturally assumes that distinctions in the state space are important. If a specification makes distinctions (two model values are unequal mathematically) without differences (variables with those two distinct values are computationally indistinguishable), confusion is inevitable. The conceptual model the specifier is trying to give the client fails to convey the true situation, and the client is likely to look for another model of the component's behavior and to translate mentally between the official specification and this alternate view [14]. The simple calculator above is a good example of this effect. If the state space is a multiset of numbers, the client is inclined to think it should be possible to use the component to find the median of the numbers entered. This client's initial expectation first will turn to confusion about the perceived incompatibility between the large state space and the limited power of the provided operations to observe it, and ultimately to disappointment that the component is not really reusable in the new situation.

1.2. The Principle of Controllability

A complementary objective to understandability is utility: a reusable component should be useful to a variety of clients whose particular needs for variants of a basic functionality are perforce unknown at component design time. Another way to view this notion of utility is in terms of "functional completeness". This suggests that the combination of operations being specified should be at least powerful enough to construct any value in the state space defined by the model.

An intuitive statement of this property is the *principle of controllability:*

C0 A specification S defining the program type ADT is controllable iff every value in ADT's state space is "computationally reachable" using some combination of operations of S.

A prime motivation for seeking controllability is technical, although it might be argued that observability is technically even more crucial. An example illustrates their combined importance. Suppose a client programmer using the specified component S wants to show that a code segment preserves the abstract value of some ADT variable. This means the segment has no net effect on the value of that variable, although the value may be changed temporarily within the segment. If S is not both observable and controllable then generally it is impossible to argue that *any* code segment does this — either because it is impossible to predictably reconstruct the original value before the end of the segment (e.g., because the original value resulted from non-deterministic behavior of some operation that is not repeatable due to lack of controllability), or because it is impossible to know that a proposed reconstructed value is really equal to the original and not simply computationally indistinguishable from it (due to lack of observability).

1.3. The Need for Practical Compliance Tests

How are observability and controllability applied in practice? Typically a designer has an informal notion of what basic functionality is sought. An initial set of operations is postulated, and the next question is what model to use to explain the state space over which these operations work. The principles of observability and controllability lead the designer to seek a state space for the specified behavior without redundant values that cluster into non-singleton congruence classes of computationally indistinguishable points, and without values that are not even reachable. A first attempt at specifying the operations is made using a "natural" model that is thought (hoped) to lead to a specification which is both observable and controllable. But sometimes it is not, in which case there are two repair strategies: try another model, or modify the behavior of some operations and perhaps add and/or remove some. In this paper we use an example that illustrates only the second approach. But in either case the designer checks again for observability and controllability. With luck, the process eventually terminates with a design that satisfies both of these design principles (and presumably others of simultaneous interest).

In order to carry out this iterative process, then, a designer has to have effective practical tests for whether a specification complies with the two principles. This requires making clear, unambiguous definitions of the principles, which is the focus of this paper.

We begin in Section 2 by reviewing related work and outlining a working example. In Section 3 we discuss ambiguities in, and possible formalizations of, **O0** and **C0**; then in Section 4 we explain how these definitions break down when applied to parameterized components that typify reusable software components (e.g., Ada generic packages and C++ class templates). Finally, in Section 5 we draw conclusions and again relate the path of this paper to the road map in Figure 1.

2. Background and Working Example

The principles of observability and controllability, as defined here, are meaningful only in the context of model-based specifications where mathematical theory and program specification are separate, as in Larch [8] and RESOLVE [3]. The question addressed by observability and controllability is essentially whether the mathematical model of an ADT is in some sense "minimal" in size and structure for specifying a programming concept. This is not a well-formed question for true algebraic specifications, in which a mathematical theory and a programming component being specified are treated as inseparable. The closely related taxonomy of mathematical functions of a theory into "observers" and "constructors" (e.g., [8, 13]) is clearly related in spirit, but these notions are one level removed as they pertain to the design of mathematical theories and not to the design of model-based specifications that use those theories.

A related issue that received much attention in the late 1970's in the algebraic specification community is when two mathematical values should be considered equal. Some authors [6, 12] considered two values to be different unless demonstrably equal based on the axioms. Others [7] considered two values to be equal unless provably different. While the first group took a traditional view and insisted that the smallest congruence relation defined by the axioms be used, the latter group allowed any congruence relations (including the smallest) consistent with the axioms. In general, for well-defined theories that are typically used as models (e.g., the Larch set trait [8]) the two notions converge. Our consideration of observability and controllability is independent of this question, because we simply assume equality in the mathematical spaces as a given predicate with the requisite properties.

The most closely related work we know about (also the most practical in terms of development of design principles) deals with "expressiveness" of the operation set of an ADT [11]. This work is similar to ours in that the authors explore a "distinguishability" relation and take a formal approach to try to minimize ambiguity in

definitions and principles. However, their specification system is algebraic, and the results apply only to immutable types and to programming operations that are total and have functional behavior. Our investigation reveals that some of the more interesting theoretical and practical questions involve relationally-defined operations and operations with non-trivial preconditions — situations that routinely arise in the design of practical reusable components. The ultimate difference between their design principles and ours is visible in our respective recommended "good" designs for a Set ADT (compare [11, page 149] with its "max" or "min" operation, and our Figure 2 with the Remove_Any operation of Section 4.2). Our design does not require an ordering on the Set elements. Our design also admits high performance implementations (e.g., hashing) that are inappropriate and ineffective with the ordering requirement. Indeed our Set ADT can be layered on any implementation of their Set ADT without a performance penalty, but not vice versa.

There are other papers dealing with issues similar to observability in other papers from the theoretical algebraic specification literature, e.g., [1]. However, the authors do not discuss implications of their work for practical design, even for algebraically-specified software components. To our knowledge, the more practical model-based specification community has not systematically considered the problem of choosing an appropriate mathematical model for specifying an ADT. There is the notion of an "unbiased" or "sufficiently abstract" or "fully abstract" model [10], which is similar to observability in the sense that it is defined almost exactly like O_0. But this informal definition leaves open the possibility of various interpretations, along the lines suggested in Sections 3 and 4. This is precisely the confusion we wish to clear up.

To illustrate these difficulties we use the example in Figure 2 of a possible specification for a Set ADT. Here the appropriate mathematical model seems clear. The question is what operations need to be provided in order to achieve observability and controllability. The specification language is RESOLVE [2, 3, 15], but the issues arise in any model-based specification language [20].

In RESOLVE, the mathematical model of an ADT is defined explicitly, as with **finite set**; or by reference to a program type, as with **math**[Item], which denotes the mathematical model type of the program type Item. Every program type in RESOLVE carries with it initialization and finalization operations (invoked in a client program through automatically-generated calls at the beginning and end of a variable's scope, respectively), and a swap operation (invoked in a client program using the infix ":=:" operator). The effect of initialization is specified in the **initialization ensures** clause. The effect of finalization usually is not specified because it has no abstract effect; in any event this aspect is unimportant here. The effect of swapping is to exchange the values of its two arguments.

Operation specifications are simplified by using abstract parameter modes **alters**, **produces**, **consumes**, and **preserves** [9]. An **alters**-mode parameter potentially is changed by executing the operation; the **ensures** clause says how. A **produces**-mode parameter gets a new value that is specified by the **ensures** clause, which may *not* involve the parameter's old value (denoted using a prefix "#") because it is irrelevant to the operation's effect. A **consumes**-mode parameter gets a new value that is an initial value for its type, but its old value *is* relevant to the operation's effect. (The rationale for using this mode for the item inserted into a Set is discussed elsewhere [9].) A **preserves**-mode parameter suffers no net change in value between the beginning of the operation and its return, although its value might be changed temporarily while the operation is executing.

The example is simple but it helps to illustrate the nature of the problems facing a specification designer. Is the specification in Figure 2 observable and controllable? What does it mean for two Set values to be "computationally distinguishable", or for a Set value to be "computationally reachable"?

```
concept Set_Template
  context
    global context
      facility Standard_Boolean_Facility
      facility Standard_Integer_Facility
    parametric context
      type Item
  interface
    type Set is modeled by finite set of
        math[Item]
      exemplar s
      initialization
        ensures  s = empty_set
    operation Insert (
        alters s: Set
        consumes x: Item)
      requires  x is not in s
      ensures   s = #s union {#x}
    operation Remove (
        alters s: Set
        preserves x: Item)
      requires  x is in s
      ensures   s = #s - {x}
    operation Is_Member (
        preserves s: Set
        preserves x: Item): Boolean
      ensures   Is_Member  iff  (x is in s)
    operation Size (
        preserves s: Set): Integer
      ensures   Size = |s|
end Set_Template
```

Figure 2 — Possible Specification of a Set ADT

3. Formalizing the Principles

In this section we consider possible interpretations of O_0 and C_0, hoping to pin down the phrases "computationally distinguishable" and "computationally reachable".

3.1. Stating the Principles More Precisely

A big problem with the informal definitions O_0 and C_0 has to do with the possibility of relationally-specified behavior. Although every operation in Figure 2 has functional behavior — the results of each operation are uniquely determined by its inputs — there are many situations where it is appropriate to define an operation so its post-condition can be satisfied in more than one possible way [19]. A correct implementation might exhibit functional behavior, but a client of the specification cannot count on any particular function being computed — only on the results of each operation satisfying the relation specified in the post-condition.

The practical difficulty this causes in applying O_0 and C_0 is that code layered on top of such a component appears to be non-deterministic, in the sense that it might do something with one implementation of the component but quite another with a different implementation. This is so even when the layered operation is specified to have functional behavior; among other things, the code implementing the layered operation might always terminate with some implementations of the underlying component, but not with others.

ψ_1 When we say "computationally distinguishable" or "computationally reachable", do we mean for *some* implementation of the component, or for *all*?

A strong version of observability is that it should be possible to write a client program that can decide equality of two variables for *every* implementation of the underlying component specification; similarly for controllability. We can formalize this by stipulating the total correctness of certain code layered on top of the specified concept. An implementation of specified behavior is totally correct if it is partially correct (i.e., correct if terminating) *and* terminating, for any totally correct implementations of the components it uses.

We select this path because it leads to the principles identified in earlier work [18], and we thereby come to the following possible formalization of observability:

O_1 A specification S defining the program type ADT is observable iff there is a totally correct layered implementation of:

```
operation Are_Equal (
    preserves x1: ADT
    preserves x2: ADT): Boolean
  ensures   Are_Equal  iff  (x1 = x2)
```

Controllability is slightly different in flavor, since as expressed in C_0 it seems to say something about an entire family of operations. It might be formalized as follows:

C_1 A specification S defining the program type ADT is controllable iff for every constant c: **math**[ADT], there is a totally correct layered implementation of:

```
operation Construct_c (
    produces x: ADT)
  ensures   x = c
```

3.2. Making the Principles Symmetric

A hint that something lurks below the surface here is the disturbing asymmetry between the definitions O_1 and C_1, the first involving a two-argument program operation and the second a quantified mathematical variable and a one-argument program operation.

ψ_2 Should observability and controllability be defined in terms of relationships between two program variables, or in terms of a program variable and a universally quantified mathematical variable, or perhaps in some other way?

Here we choose the first path, which we took in deriving the principles published earlier [18] and which *a priori* seems as reasonable as any other. The revision needed for controllability, however, makes it clear that the definition is contingent, or relative, in the following sense. "Computationally reachable" does not mean (as in C_1) that every value in the state space can be constructed from scratch, i.e., starting from an initial value of the ADT. It means that every value in the state space can be reached from every other — even if the given starting point could not itself have been constructed from scratch. The meaning of C_2 is now apparently quite different from that of C_1, which is an "absolute" notion of controllability in that there is only one variable involved. So we add the modifier "relatively" in defining both principles as follows:

O_2 A specification S defining the program type ADT is *relatively* observable iff there is a totally correct layered implementation of:

```
operation Are_Equal (
    preserves x1: ADT
    preserves x2: ADT): Boolean
  ensures   Are_Equal  iff  (x1 = x2)
```

C_2 A specification S defining the program type ADT is *relatively* controllable iff there is a totally correct layered implementation of:

```
operation Get_Replica (
    preserves x1: ADT
    produces x2: ADT)
  ensures   x2 = x1
```

These definitions match practical compliance tests of prior work [18]. But they still have some technical problems, which we explore next.

3.3. Making the Principles More Independent

By definitions O_2 and C_2, relative observability is not entirely independent of relative controllability, since it demands that the arguments to Are_Equal should be preserved and this apparently requires some degree of controllability. Similarly, the first argument to Get_Replica must be preserved and proving this seemingly requires observability, as noted in Section 1.2. Is it possible to define the principles so they are not so evidently connected? The heart of the problem is that both definitions O_2 and C_2 involve preservation of operation arguments. We are, therefore, led to consider this variation:

O_3 A specification S defining the program type ADT is relatively observable iff there is a totally correct layered implementation of:

```
operation Were_Equal (
    alters x1: ADT
    alters x2: ADT): Boolean
  ensures  Were_Equal iff (#x1 = #x2)
```

This definition is a bit curious because, technically in RESOLVE, a function operation may have only **preserves**-mode parameters; but a violation here seems justifiable for ease of explanation. The parallel definition for relative controllability is:

C_3 A specification S defining the program type ADT is relatively controllable iff there is a totally correct layered implementation of:

```
operation Move (
    alters x1: ADT
    produces x2: ADT)
  ensures  x2 = #x1
```

3.4. Relationships Among the Above Definitions

Definitions O_3 and C_3 make the principles no stronger than with definitions O_2 and C_2, in the sense that any specification that is relatively observable (controllable) by O_2 (respectively, C_2) is equally so by O_3 (respectively, C_3). The reason is that it is trivial to layer an implementation of Were_Equal (Move) on top of Are_Equal (respectively, Get_Replica). Furthermore, if a specification is relatively observable by definition O_3 and relatively controllable by definition C_2, then it is relatively observable by definition O_2 because we can layer Are_Equal on top of Get_Replica and Were_Equal:

```
operation Are_Equal (
    preserves x1: ADT
    preserves x2: ADT): Boolean
  local context
    variables copy1, copy2: ADT
  begin
    Get_Replica (x1, copy1)
    Get_Replica (x2, copy2)
    return Were_Equal (copy1, copy2)
  end Are_Equal
```

Also note that every RESOLVE specification is relatively controllable by definition C_3, since every type comes with swapping. Here is a universal implementation of Move in RESOLVE:

```
operation Move (
    alters x1: ADT
    produces x2: ADT)
  begin
    x1 :=: x2
  end Move
```

In effect, a move is half a swap. This is one reason we previously suggested the guideline of testing the stronger criteria O_2 and C_2 [18]. For components in other languages, however, C_3 is a non-trivial criterion. For example, consider an Ada package defining a Stack ADT as a limited private type (no assignment operator), along with operations Push, Pop, and Is_Empty having the usual meanings. This is relatively controllable by C_3 — but not because a primitive data movement operator for Stacks is trivially assumed. Without any one of the three operations it would *not* be relatively controllable by C_3.

The relationships among the definitions in this section are depicted in the Venn diagram of Figure 3, where we take the liberty of labeling sets of specifications with the labels of the definitions under which their member specifications qualify.

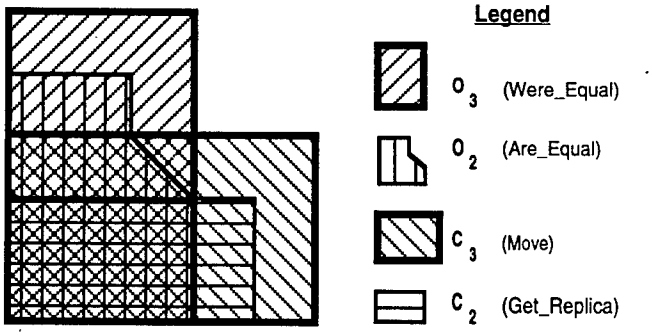


Figure 3 — Relationships Among Definitions

4. Parameterized Components

At first the above definitions seem clear and unambiguous. But suppose we try to apply those definitions to the Set_Template specification of Figure 2. It seems the specification in Figure 2 should be deemed *not* observable by O_0 because there is no practical way to enumerate the elements of a Set, and this should be crucial in computationally distinguishing between two unequal Sets. It seems the specification should be deemed controllable by C_0, however, because starting from an empty set it is easy to construct any finite set by repeated Inserts. Does this intuition match what the proposed definitions say? We discuss in detail only O_3, considerations for the other definitions being similar.

4.1. Type Parameters and Modular Proofs

There is a reasonable way to interpret **O3** that makes the Set_Template specification observable. The key features that permit this view are that **O3** defines relative observability in terms of the existence, not the practicality, of an implementation of Were_Equal; and that there is no restriction on the assumptions an implementer of Were_Equal may make about the available operations on Items.

We start by noting that the mandated existence of "a totally correct layered implementation" of the Were_Equal operation for Set_Template means, in RESOLVE terms, the existence of a totally correct implementation of the following concept:

```
concept Set_Were_Equal_Capability
  context
    global context
      facility Standard_Boolean_Facility
      concept Set_Template
    parametric context
      type Item
      facility Set_Facility is
        Set_Template (Item)
  interface
    operation Were_Equal (
        alters s1: Set
        alters s2: Set): Boolean
      ensures  Were_Equal  iff  (#s1 = #s2)
end Set_Were_Equal_Capability
```

This formulation makes clear that the implementation of Were_Equal must be layered, since an instance of Set_Template is a parameter to the concept. Moreover, it makes clear that the implementation must work for any type Item for the Set elements, since Item also is a parameter. What it does not make clear, however, is what other components and services an implementation might use and depend on.

In the absence of restrictions, presumably any such services may be assumed — a rather liberal interpretation of **O3**. But now what prevents an implementer of Were_Equal from simply assuming the existence of a (possibly thinly disguised) operation that tests equality of Sets of Items, and layering on top of that? Nothing.

So we might wish to use a less liberal interpretation of **O3**. For example, suppose we insist that an allowable implementation of Were_Equal may not use any operations with Set parameters other than those from Set_Template itself. Unfortunately, this does not solve the problem either. For example, below is a possible algorithm for Were_Equal, which is built on top of Set_Template and an "enumerator" concept for Items. In RESOLVE's modular proof system, total correctness is defined in such a way that the following code is a totally correct implementation of Were_Equal, because we *assume* there is a totally correct implementation of the enumerator interface and the total correctness of the Set_Template implementation — and because all Sets are finite. As a result we claim that Set_Template is relatively observable even by this less liberal interpretation of **O3**.

```
operation Were_Equal (
    alters s1: Set
    alters s2: Set): Boolean
  local context
    variables x: Item
  begin
    if (Size (s1) = 0 and Size (s2) = 0)
      then return true
      else
        let x = any Item value not
          previously enumerated during the
           top level call of Were_Equal
        if Is_Member (s1, x)
          then
            if Is_Member (s2, x)
             then
                Remove (s1, x)
                Remove (s2, x)
                return Were_Equal (s1, s2)
              else return false
            end if
          else
            if Is_Member (s2, x)
              then return false
              else return Were_Equal (s1,s2)
            end if
        end if
    end if
  end Were_Equal
```

This illustrates the power of a modular proof system [5]. There might be Items for which it is impossible to implement the enumerator interface, but this does not influence the total correctness of Were_Equal. At the mathematical level, if the state space **math**[Item] is effectively enumerable then in principle there exists an implementation of the enumerator interface. But only if the specification of the actual program type Item is at least controllable, by a reasonable definition, should we expect to be able to implement the enumerator interface for it.

So perhaps we should insist that the underlying components actually should be implementable. But then should the mere possibility of instantiating Set_Were_-Equal_Capability with an Item for which the enumerator cannot be implemented be enough to render the Set_Template specification not observable? And does "possibility" here mean the library of components *actually contains* such a type, or that in principle it *might contain* such a type? Suppose, for example, that in the specification language it is simply impossible to specify a program type whose state space is not enumerable. Should this situation — which might be reasonably attributed to inexpressiveness of the specification language and not to a problem with the design of Set_-

Template — be the deciding factor as we attempt to apply the observability test to Set_Template?

If we use an interpretation in which the above implementation of Were_Equal is acceptable, so Set_Template is deemed relatively observable, then it is interesting to see where variants of Set_Template lie in Figure 3. In Figure 4, we have placed some of them to illustrate the limited discriminating power of the definitions. For example, Get_Replica for Sets can be layered on top of Are_Equal for Sets using only Swap and Insert: systematically generate candidate Sets by enumerating Items and inserting them into empty Sets — first one Set with one Item, then two Sets with one Item and two Sets with two Items, and so forth — stopping when the Set to be copied and the current candidate Are_Equal. There is no need for Remove, Is_Member, or Size.

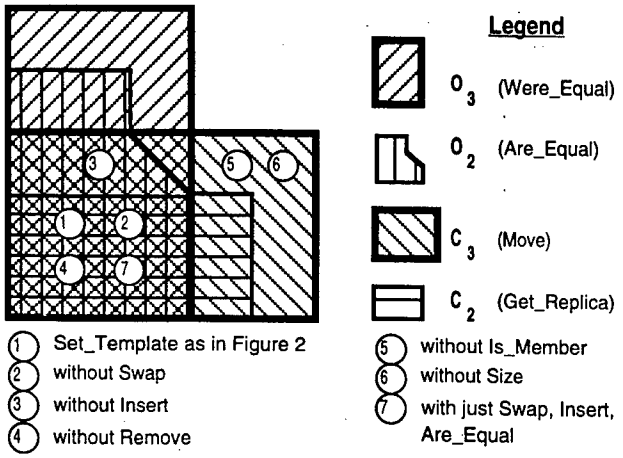


Figure 4 — Variants of Set_Template Assuming math[Item] is Enumerable

It should be clear that these definitions are not really "right", in the sense that even if they do capture some sense of observability and controllability they do not rule out patently poor specifications. For example, Set_Template itself (even without Swap) is both relatively observable and relatively controllable by the strong definitions O_2 and C_2, despite providing no practical way to enumerate the elements of a Set. Even Set_Template without Remove is relatively observable and relatively controllable, as it is with just Swap, Insert, and Are_Equal.

4.2. Handling Parameterized Components

The difficulties in Section 4.1 are traceable to the prospect of having specifications that are parameterized by another type Item, and to the absence of restrictions on the assumptions an implementation may make about the actual Item type. Even allowing an implementation of Were_Equal to rely only on the assumption that the state space of Item is enumerable weakens the definitions so much that they are practically worthless.

Some features of RESOLVE permit us to easily clarify and strengthen the previous definitions to deal with parameterized modules, so the observability of a parameterized type is unaffected by properties of the arbitrary type by which it is parameterized. Each realization (implementation) of a concept may require additional parameters beyond those of the concept, and these appear in the realization "header" [2]. This mechanism lets us require that the implementation of an operation Were_Equal for type Set may only count on the always-present initialization, finalization, and swapping for Items, *and* on a similarly-defined Items_Were_Equal operation. Any allowable realization of the concept exporting Were_Equal should have a realization header in which this one operation is the only realization parameter.

This leads to a refined definition of relative observability (the others being similar):

O_3' A specification S, parameterized by the program type Item and defining the program type ADT, is relatively observable iff there is a totally correct implementation of:

```
concept S_Were_Equal_Capability
  context
    global context
      facility Standard_Boolean_Facility
      concept S
    parametric context
      type Item
      facility S_Facility is S (Item)
  interface
    operation Were_Equal (
        alters x1: ADT
        alters x2: ADT): Boolean
      ensures   Were_Equal  iff
                  (#x1 = #x2)
end S_Were_Equal_Capability
```

whose realization context makes only the following additional mention of Item:

```
realization header Allowed
    for S_Were_Equal_Capability
  context
    parametric context
      operation Items_Were_Equal (
          alters x1: Item
          alters x2: Item): Boolean
        ensures   Were_Equal  iff
                    (#x1 = #x2)
end Allowed
```

In applying this definition to Set_Template, we find there is no way for the realization body of Set_Were_Equal_Capability to use any externally-provided operations involving Items, other than Items_Were_Equal. This rules out impractical but technically correct implementations like the one in Section 4.1.

Figure 5 is the counterpart of Figure 4, with the refined definitions. Now Set_Template is not relatively observable by $O_{2'}$ or by $O_{3'}$, nor relatively controllable by $C_{2'}$. However, by adding the following operation (or something similar) it becomes relatively observable and relatively controllable even by $O_{2'}$ and $C_{2'}$:

```
operation Remove_Any (
    alters s: Set
    produces x: Item)
  requires  s /= empty_set
  ensures   (x is in #s) and (s = #s - {x})
```

Remove_Any (s, x) removes an arbitrary element of the original s and returns it in x. Now there is a practical way to enumerate the elements of a Set, leading to obvious implementations of the required layered operations that assume no more than the ability to do with Items what the layered operation is doing to Sets.

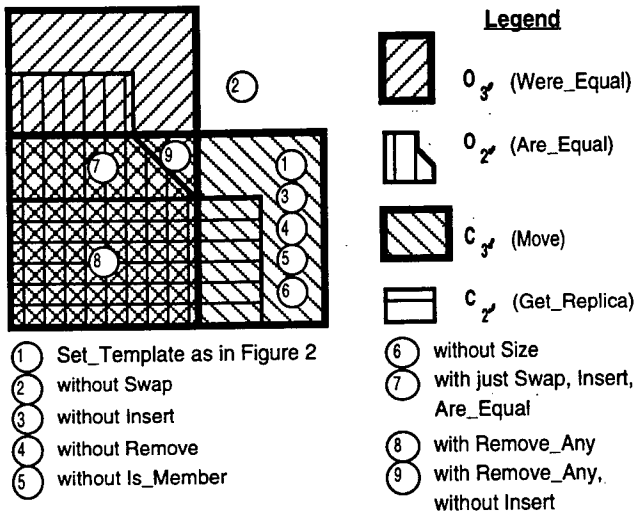


Figure 5 — Variants of Set_Template With Section 4 Definitions

Figure 5 shows what happens to the variants of Set_Template previously displayed in Figure 4 (circles 1-7). Two new variants help to illustrate the discrimination power of the new definitions. Set_Template with Remove_Any (circle 8) — a good design — passes both of the stronger compliance tests $O_{2'}$ and $C_{2'}$. Set_-Template with Remove_Any but without Insert (circle 9) — plainly not a good design — still passes both weaker tests $O_{3'}$ and $C_{3'}$ but neither stronger one. So the definitions used for Figure 5 seem better than those used for Figure 4. But again even $O_{2'}$ and $C_{2'}$ clearly are not "right" in that they still do not rule out patently bad specifications. It is easy to circumvent their intent by attacking the symptoms and not the disease: just add Are_Equal and Get_Replica as primary operations. In fact, Set_Template with just Are_Equal and Get_Replica and no other operations whatsoever sits in precisely the same place in Figure 5 as Set_Template with Remove_Any, despite clearly not satisfying C_1. Fixing these problems apparently requires taking a different path altogether, as we discuss in the conclusions below.

5. Conclusions

A fundamental question facing the designer of a model-based specification of an ADT is the appropriateness of the chosen conceptual model. We have discussed some of the technical problems in carefully defining two principles that provide the specifier with criteria for appropriateness: Does the chosen model interact with the specified operations in a way that makes the specification observable and controllable? A negative answer on either count suggests that the specifier needs to look harder, or be prepared to justify non-compliance on the basis of other requirements. A positive answer on both counts gives a certain confidence, though among satisfactory specifications some may be "better" than others (e.g., more understandable or more flexible). However, it hardly guarantees that the specification is "good" in any reasonable and absolute intuitive sense.

We mentioned alternate paths that might be followed to formalize observability and controllability. Here are some conclusions from preliminary exploration of these paths—conclusions not justified in the body of this paper.

Ψ_1 When we say "computationally distinguishable" or "computationally reachable", do we mean for *some* implementation of the specified component, or for *all*?

Defining the principles using an existential quantifier over implementations is largely unexplored territory. However, there is reason to believe it might be attractive. Consider, for example, the specification of an ADT called Computational_Real modeled as a real number. The operations have relationally-defined behavior. The Add operation, for example, ensures that the result of adding two Computational_Reals is a Computational_Real whose model lies within some small interval around the sum of the models of the addends. Based on a cardinality argument, it is clear there is no way the specification can be deemed controllable if we insist that *every* implementation of it *must* support reaching every real number. However, the obvious Computational_Real operations (which mirror the usual mathematical operators for reals) are powerful enough to allow that every real number *might* be reachable in *some* implementation, since the union of the allowed intervals over all computations with these operations just has to cover the reals. The power of relationally-specified behavior is evident here, but the full implications of defining observability and controllability as suggested are not.

ψ_2 Should observability and controllability be defined in terms of relationships between two program variables ("relatively"), or in terms of a program variable and a universally quantified mathematical variable, or perhaps in some other way?

Defining both principles the second way leads to interesting phenomena and to other interesting questions involving the expressiveness of the mathematics and the relationships between those definitions and the ones in this paper. Observability basically becomes a test of whether, for every point in the state space, it is possible to tell whether a program variable Was_Equal to it. Controllability is more properly termed "constructability", using something like definition $\mathbf{C_1}$. These alternate definitions cut through diagrams like Figures 3-5 in a surprising way, since there are specifications that are observable and/or controllable by the alternate definitions but not by $\mathbf{O_2}$' and/or $\mathbf{C_2}$', and vice versa. So such definitions might offer distinct useful tests which should be applied in tandem with the ones described here, when evaluating a proposed specification.

6. Acknowledgment

We thank Murali Sitaraman and Stu Zweben for insightful comments on a draft of this paper, and the anonymous referees for their helpful suggestions and pointers to some relevant literature (especially [11]). We also gratefully acknowledge financial support for our research from the National Science Foundation under grant CCR-9311702, and from the Advanced Research Projects Agency of the Department of Defense under ARPA contract number F30602-93-C-0243, monitored by the USAF Materiel Command, Rome Laboratories, ARPA order number A714.

7. References

[1] Bernot, G., Bidoit, M., and Knapik, T., "Observational Specifications and the Indistinguishability Assumption," Theoretical Computer Science 139, 1995, 275-314.

[2] Bucci, P., Hollingsworth, J.E., Krone, J., and Weide, B.W., "Implementing Components in RESOLVE," *Software Engineering Notes 19*, 4, October 1994, 40-52.

[3] Edwards, S.H., Heym, W.D., Long, T.J., Sitaraman, M., Weide, B.W., "Specifying Components in RESOLVE," *Software Engineering Notes 19*, 4, October 1994, 29-39.

[4] Edwards, S.H., *A Formal Model of Software Subsystems*, Ph.D. dissertation, Dept. of Computer and Information Science, The Ohio State Univ., Columbus, March 1995.

[5] Ernst, G.W., Hookway, R.J., and Ogden, W.F., "Modular Verification of Data Abstractions with Shared Realizations," *IEEE Transactions on Software Engineering 20*, 4, April 1994, 288-307.

[6] Goguen, J.A., Thatcher, J.W., and Wagner, E.G., "An Initial Algebra Approach to the Specification, Correctness, and Implementation of Abstract Data Types," in *Current Trends in Programing Methodology 4*, R. T. Yeh, ed., Prentice-Hall, 1978, 80-149.

[7] Guttag, J.V., Horowitz, E., and Musser, D.R., "Abstract Data Types and Software Validation," *Communications of the ACM 21*, 12, December 1978, 1048-1064.

[8] Guttag, J.V., and Horning J.J., *Larch: Languages and Tools for Formal Specification*, Springer-Verlag, 1993.

[9] Harms, D.E., and Weide, B.W., "Swapping and Copying: Influences on the Design of Reusable Software Components," *IEEE Transactions on Software Engineering 17*, 5, May 1991, 424-435.

[10] Jones, C.B., *Systematic Software Development Using VDM*, 2nd ed., Prentice-Hall, 1990.

[11] Kapur, D., and Mandayam, S., "Expressiveness of the Operation Set of a Data Abstraction," in *Conference Record 7th Annual Symposium on Principles of Programming Languages*, ACM, 1980, 139-153.

[12] Liskov, B.H., and Zilles, S.N., "Specification Techniques for Data Abstractions," *IEEE Transactions on Software Engineering SE-1*, 1, March 1975, 7-19.

[13] Liskov, B., and Guttag, J., *Abstraction and Specification in Program Development*, McGraw-Hill, 1986.

[14] Norman, D.A., *The Design of Everyday Things*, Doubleday/Currency, 1990.

[15] Ogden, W.F., Sitaraman, M., Weide, B.W., and Zweben, S.H., "The RESOLVE Framework and Discipline — A Research Synopsis," *Software Engineering Notes 19*, 4, October 1994, 23-28.

[16] Sitaraman, M., Harms, D.E., and Welch, L.W., "On Specification of Reusable Software Components," *International Journal of Software Engineering and Knowledge Engineering 3*, 2, June 1993, 207-229.

[17] Spivey, J.M., *The Z Notation: A Reference Manual*, Prentice-Hall, 1989.

[18] Weide, B.W., Ogden, W.F., and Zweben, S.H., "Reusable Software Components", in *Advances in Computers, vol. 33*, M.C. Yovits, ed., Academic Press, 1991, 1-65.

[19] Weide, B.W., Ogden, W.F., and Sitaraman, M., "Recasting Algorithms to Encourage Reuse," *IEEE Software 11*, 5, September 1994, 80-88.

[20] Wing, J.M., "A Specifier's Introduction to Formal Methods", *Computer 23*, 9, September 1990, 8-24.

Representation Inheritance: A Safe Form of "White Box" Code Inheritance

Stephen H. Edwards, *Member, IEEE Computer Society*

Abstract—There are two approaches to using code inheritance for defining new component implementations in terms of existing implementations. Black box code inheritance allows subclasses to reuse superclass implementations as-is, without direct access to their internals. Alternatively, white box code inheritance allows subclasses to have direct access to superclass implementation details, which may be necessary for the efficiency of some subclass operations and to prevent unnecessary duplication of code.

Unfortunately, white box code inheritance violates the protection that encapsulation affords superclasses, opening up the possibility of a subclass interfering with the correct operation of its superclass' methods. *Representation inheritance* is proposed as a restricted form of white box code inheritance where subclasses have direct access to superclass implementation details, but are required to respect the representation invariant(s) and abstraction relation(s) of their ancestor(s). This preserves the protection that encapsulation provides, while allowing the freedom of access that component implementers sometimes desire.

Index Terms—Abstraction function, abstraction relation, behavioral subtype, inheritance, model-based specification, object-oriented, representation invariant, reuse, specialization, subclass.

1 Introduction

Conventional wisdom about how best to use inheritance in object-oriented (OO) programming often centers around the reasoning problems of component clients, not implementers. Most solutions, e.g., adherence to the Liskov Substitutability Principle (LSP) [1], helpfully instruct component designers in the correct way to use specification inheritance. Unfortunately, these solutions do not address the code reuse problems that also affect class designers.

Specifically, code inheritance that allows a subclass to directly access the representation it inherits from its parent—which we might consider *white box* code inheritance—raises serious concerns about safety, correctness, and loss of locality when reasoning about implementations. In contrast, with *black box* code inheritance new features in a subclass are simply additions that are written in terms of the superclass' external client interface. Fig. 1 illustrates these two approaches, where a subclass of a basic list abstraction adds a `Reverse()` operation to the behavior it inherits from its parent. This paper addresses the utility of white box code inheritance as a practical mechanism for component implementers, describes the drawbacks it entails and their theoretical roots, and proposes *representation inheritance*—a *safe* variety of white box code inheritance that meets practical needs without raising the same concerns.

Section 2 explains why problems arise from white box code inheritance, and defines representation inheritance. Section 3 elaborates the discussion of the problems of white box code inheritance through a simple example—a two-way list component. Section 4 then shows how representation inheritance can be applied in the example. Section 5 comments on methods of enforcing the restrictions imposed by representation inheritance, and finally Section 6 discusses relationships with previous work.

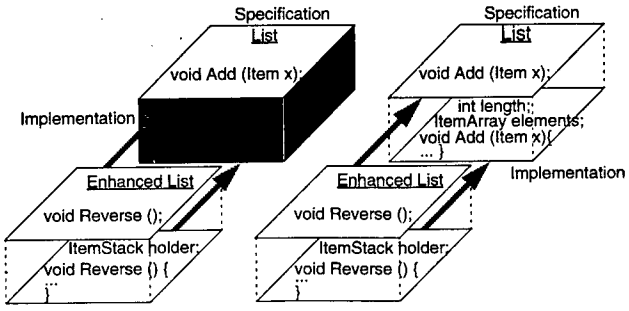


Fig. 1. Black box vs. white box inheritance.

2 The Problem

When defining a new subclass, an OO programmer often has the option of implementing some or all of a subclass' new features by directly manipulating the data members and/or using the internal operations inherited from its superclass,[1] which we call white box code inheritance. Unfortunately, a subclass implemented using white box code inheritance has a "back door" through the protection that encapsulation normally affords its parent. This leak opens the possibility of subclass code compromising the integrity of an encapsulated object's internal representation. As a result, one can no longer reason about the behavior of a particular method just by

• *S.H. Edwards is with the Department of Computer Science, Virginia Polytechnic Institute and State University, Blacksburg, VA 24061. E-mail: edwards@cs.vt.edu.*

Manuscript received June 26, 1996; revised Nov. 11, 1996.
Recommended for acceptance by S.H. Zweben and M. Sitaraman.
For information on obtaining reprints of this article, please send e-mail to: transse@computer.org, and reference IEEECS Log Number S97026.

1. The problems and solutions we discuss may involve either single or multiple inheritance, but the descriptions in this paper are written in terms of single inheritance for simplicity.

looking at the class it is in—any present *or future* subclass has the potential to interact with it indirectly through the object's internal state in unforeseen ways.

For these reasons, many researchers and practitioners alike have advocated avoiding white box code inheritance completely. Why then would a programmer ever choose to use it? There are two reasons for using white box code inheritance in practical situations. The more commonly cited, but less compelling, reason is efficiency. In some circumstances, subclass operations implemented using black box techniques suffer large space or time performance penalties that could be avoided through the use of white-box code inheritance, as in the example component described in Section 3. When such a case arises, one might suggest simply reimplementing the new class independently, perhaps as a sibling rather than as a descendant of the chosen superclass.

Unfortunately, this leads to the less commonly cited but more compelling reason for using white box code inheritance—avoiding the extra testing and maintenance burden required by duplicating code. The "cut-and-paste"-style reuse involved in reimplementing a class separately in order to add a new method that requires direct access may save coding time, but provides no help for testing or maintenance—the two classes will require twice as much effort as the original class alone [2]. Ideally, one would instead like to consider the newly added code in a subclass to be independent of any inherited code for the purposes of testing and maintenance. While unrestricted white box code inheritance does not admit this possibility, representation inheritance does, as explained in Section 5.

For the remainder of this section, we turn our attention to the hole in class encapsulation that white box code inheritance opens. The difficulties that arise from this leak occur when a subclass either fails to respect its parent's representation invariant, or fails to respect its parent's abstraction relation.

2.1 Respecting Representation Invariants

Internally, an object's methods interact indirectly with each other through the state variables the object encapsulates. Because this interaction is indirect, its success critically depends on assumptions about the meanings attributed to the variables and to changes in their values—assumptions shared by all the methods. A class' *representation invariant* captures exactly these assumptions [3, pp. 72-74].

As an example, consider a class that implements the abstraction of a "list of items." As shown in Fig. 1, one way to implement such a class is to give it two internal state variables: an array of items called "`elements`," and an integer called "`length`" recording how much of the array is in use. One might design the methods for this class so the value of `length` always refers to some valid index into the `elements` array—a representation invariant which all of the class methods would share.

Typical OOPLs encourage one to encapsulate object state information within a class so that clients cannot violate assumptions that are critical to the correct functioning of the class' methods. However, subclasses may occasionally need direct access to a superclass' internal state (i.e., the specialization interface may provide a different view of the class than the client interface). This access allows them to manipulate that representation in ways that can violate the representation invariant, introducing "bug-like" behavior in previously correct superclass methods.

To preclude the problems this unchecked freedom can introduce, we propose that:

> If a subclass has direct access to the internal state of a superclass, it is likewise obliged to live by and uphold the common assumptions shared by all methods that have direct access to those internal details—e.g., the superclass' representation invariant.

2.2 Respecting Abstraction Relations

A class' client interface is often expressed at a different level of abstraction from its internal representation details (for example, a list described to the client as a mathematical string or sequence, but represented as a linked list of nodes). The correspondence between the internal state representation of an object and its intended conceptual value is expressed as an *abstraction function* [3, pp. 70–71] or, more generally, *abstraction relation* [4], [5].

Again as an example, consider our "list of items" abstraction in Fig. 1. One might assume that the items stored in the list are recorded in the `elements` array, while the `length` state variable records how much of the array is in use. Even so, there are still a variety of alternatives for representing the list's conceptual value in these variables. Which series of contiguous items in the `elements` array form the list: those before `length`, or those after? Does the `length` state variable indicate the index of the last item of the list, or does it refer to the first unused array index *after* the list? These choices are part of the abstraction relation for this class.

Subclasses that are intended to be behavioral subtypes of their superclasses must obey the Liskov Substitutability Principle, meaning that at the level of abstraction in the client interface, objects of the subclass must behave in a manner consistent with the superclass. To ensure that behavioral subtypes behave consistently, in addition to the LSP we propose that:

> If a subclass is intended to be a behavioral subtype, yet has direct access to the representation of its superclass, it must live by and uphold the abstraction relation shared by all the methods that have direct access to those internal details. Behavioral substitutability (in the LSP sense) must also be established for any internal superclass methods that are overridden.

2.3 Representation Inheritance

Representation inheritance is a term for code inheritance where a subclass has white-box access to its parent's internals, *and* the subclass respects the parent's representation invariant. Because most modern OO programming languages (OOPLs) provide only one inheritance mechanism, when behavioral subtyping is desired we will consider representation inheritance to encompass the requirement for a subclass to respect both superclass invariants and superclass abstraction relations.

Representation inheritance is built on lessons learned from model-based specification techniques [6], [7], which require one to explicitly state representation invariants and abstraction relations. This solution is notably different from

other proposed solutions, in that it does not involve partitioning a class into groups of interdependent methods that must be considered together when specializing the class. Lamping's work [8], [9], as well as that of Stata and Guttag [10], both indirectly address the difficulties of white-box reuse by grouping methods that depend on common assumptions, signaling to the specializer that these groups need to be examined or changed together. Here, we instead focus directly on the root of the problem—the shared (but often undocumented) assumptions upon which these methods depend. By capturing these assumptions in a representation invariant, it is possible to treat all inherited methods uniformly and independently, while simultaneously documenting exactly the assumptions about state maintenance upon which they depend.

3 An Example: A Two-Way List Component

3.1 The Two-Way List Abstraction

To ground the discussion of code inheritance, consider a class implementing the abstract notion of a "two-way list." Conceptually, the value of a list object is simply a sequence of items that we can visualize as being arranged in a row from left to right. Without loss of generality, consider the left end of the row to be the front or head of the list, and the right end to be the back. As we advance down the list, we can imagine that there is also a "fence" separating the items we have already seen from those that lie ahead—it partitions the row by sitting between two items. This particular list component is "two-way" because we wish to be able to move either left or right in the sequence of items.

One simple formalization of this model of two-way lists is:

```
type Two_Way_List is modeled by (
        left  : string of math [Item],
        right : string of math [Item]
    )
    exemplar l
    initialization
ensures     l.left = empty_string and
            l.right = empty_string
```

This formalization uses the notation of RESOLVE [11], although any convenient model-based specification notation could be used [6]. In this formal model of the type, the notion of the "fence" dividing the list of items into two halves is implicit: The value of a list is modeled as two separate "strings" (or sequences) that model the two parts of the row of items to the "left" and "right" of the fence.

With this model in mind, it is possible to decide on the basic operations for two-way lists. The basic operations provided for two-way list objects, as adapted from [7], include:

Move_To_Start (). Moves the fence to the beginning (left) of the list.

Move_To_Finish (), Moves the fence to the end (right) of the list.

Advance (). Moves the fence one position forward (right).

Retreat (). Moves the fence one position backward (left).

Add_Right (x). Adds x to the list right after (to the right of) the fence, and returns with x having an initial value for its type.

Remove_Right (x). Removes the item immediately following (to the right of) the fence, and returns it in x.

At_Start (). Returns true when the fence is at the far left end of the list.

At_Finish (). Returns true when the fence is at the far right end of the list.

In an object-oriented programming language such as C++, we might declare a class realizing this abstract concept as shown in Fig. 2. The C++ Two_Way_List class template in Fig. 2 defines a generic component that is parameterized by the type of item in the list. It is similar in several respects to Bertrand Meyer's ***BILINEAR*** [12, pp. 141–146] and ***TWO_WAY_LIST*** [12, pp. 154–155, 299–303] components, although Meyer's selection of primary operations and conceptual model differs in several details.

```
template <class Item>
class Two_Way_List
{
private:
    // Prevent assignment
    Two_Way_List& operator = (
        const Two_Way_List& rhs);
    // Prevent copy construction
    Two_Way_List (const Two_Way_List& 1);

public:
    // The external interface

    Two_Way_List ();
    ~Two_Way_List ();

    void      Move_To_Start  ();
    void      Move_To_Finish ();
    void      Advance        ();
    void      Retreat        ();
    void      Add_Right      (Item& item);
    void      Remove_Right   (Item& item);
    void      Remove_Right   (Item& item);
Boolean  At_Start       ();
Boolean  At_Finish      ();
};
```

Fig. 2. A C++ two-way list class template.

3.2 Implementing Two-Way List

Given this Two_Way_List declaration, we can now turn our attention to how one might implement a two-way list. For the purposes of this paper, the sample implementation will be presented in C++, although any other OO programming language could be used.

One obvious way to implement the Two_Way_List component appears in many data structures text books: Use a doubly-linked chain of nodes, where the links are implemented using pointers. A technique that can help with this approach is the use of sentinel nodes. By placing sentinel nodes (or "dummy" nodes)[2] at either end of the chain, all list operations can be handled uniformly—there are no special cases to handle when operating on either end of the chain.

2. Joe Hollingsworth, in a private communication, noted he regularly uses the approach of dual sentinel nodes when teaching linked representations to his CS1/CS2 students at Indiana University Southeast. Because of the subsequent notable reduction in bugs in student programs, he refers to such sentinels as "smart" nodes.

Given that the sequence of items contained within a Two_Way_List object will be held in a doubly-linked chain, only one question remains: What is the exact representation of a Two_Way_List object? One obvious choice is to represent a Two_Way_List by a pair of pointers: one to record the location of the head of the list, and another to record the location of the fence. Here, we arbitrarily choose for a two-way list object to have two data members, a pre_front pointer that points to the sentinel node at the front end of the chain, and a pre_fence pointer that points to the node containing the item immediately preceding the fence. Many other combinations would work just as well. This choice is elaborated in Figs. 3 and 4.

```
template <class Item>
class Two_Way_List
{

    // ...
    // The same external declarations as in
    // Figure 3
    // ...

private:
    // The representation of the class:
    struct TWL_Node          // A two-way list node
    {
      Item i;
      TWL_Node* next;        // forward pointer down
                             //    the list
      TWL_Node* previous;    // backward pointer up
                             //    the list
    };
    TWL_Node* pre-front;     // pointer to first
                             //    sentinel
    TWL_Node* pre_fence;     // pointer to node just
                             // before the "fence"
};
```

Fig. 3. The representation of Two_Way_List.

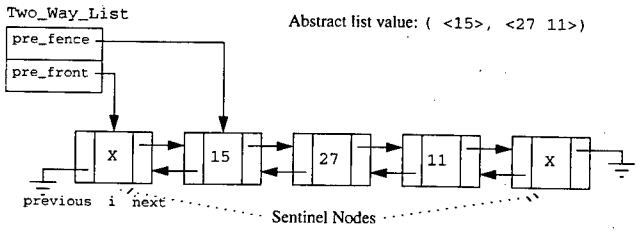


Fig. 4. A doubly-linked chain with sentinel nodes.

Fig. 3 shows the remainder of the Two_Way_List C++ class template declaration, including the declaration of the internal TWL_Node struct and of the data members holding the pre_front and pre_fence pointers. Fig. 4 then gives a pictorial representation of an actual Two_Way_List object where the items are integers. The sample list chosen has three items in the list (15, 27, and 11), with the fence currently between the first and second elements (after 15 and before 27). Fig. 4 also shows the corresponding abstract value of such a list in terms of the model defined above. Now that the representation choices have been made, providing code for the class methods is a straightforward process that is skipped here.

3.3 An Enhancement

Now that we have an example class component defined and implemented, we can turn our attention to a typical programming task: How can we extend this component with new operations that provide additional capabilities? For the purposes of this paper, we'll restrict ourselves to a simple extension: the addition of an operation called Swap_Rights that exchanges the tails (or right halves) of the two lists involved. Fig. 5 shows the Enhanced_Two_Way_List class template that adds the new method. Fig. 5 also shows a post-condition describing the behavior of the Swap_Rights operation in terms of the type's abstract model, using "#" to denote the value of an object before the method invocation.

Fig. 6 more concretely illustrates the effect of the Swap_Rights operation on two lists, where the items are integers. The first list has three elements, 15, 27, and 11, with its fence located between the first and second items. The second has four elements, 14, 87, 9, and 12, with the fence between the second and third items. Fig. 6 shows the effect of invoking the Swap_Rights method of the first list, passing the second list as the "rhs" argument to the operation. The sequences of items to the right of the fence in each list are exchanged.

The Swap_Rights operation is an interesting additional capability for two-way lists. Using the primary operations shown in Fig. 2, the only way to combine two lists, or separate one list into parts, is through a series of individual add and remove operations. The Swap_Rights operation is a useful building block that greatly simplifies the implementation of higher-level operations like concatenation, splitting, splicing, etc. Given the implementation for Two_Way_List based on sentinel nodes, what is the safest and most effective way of implementing the Swap_Rights operation?

```
template <class Item>
class Enhanced_Two_Way_List : public
Two_Way_List<Item>
{
public:
    void Swap_Rights (
        Enhanced_Two_Way_List& rhs);
        //ensures self =
        //          (#self.left, #rhs.right)
        //        and rhs =
        //          (#rhs.left, #self.right)
}
```

Fig. 5. The Enhanced_Two_Way_List class.

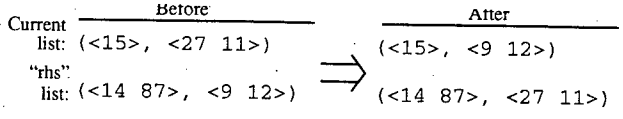


Fig. 6. The effect of Swap_Rights.

3.4 Implementing the Enhancement

Note that Enhanced_Two_Way_List can be implemented *without* access to the internals of its superclass. By treating the superclass as a black box, Swap_Rights could be implemented by moving each item from the right half of the first list over to the second list, one at a time. Then the items

from the right half of the second list have to be moved over to the first, one at a time. This will take time proportional to the number of items in the right halves of both lists.

Clearly, black box code reuse is safe, since subclasses have no more privileges than other clients when it comes to the internal representation of a superclass. One might even be tempted to claim that all code reuse should be achieved through black box methods. Unfortunately, the `Two_Way_List` example illustrates why implementers still turn to white box techniques for some problems.

`Two_Way_List`'s doubly-linked chain representation lends itself to a much more efficient (and less complex!) implementation of `Swap_Rights`. It is only necessary to change two pointer values in each chain in order to exchange the right halves of the two lists, resulting in a constant-time implementation of the operation. Doing this requires access to the representation of the `Two_Way_List` superclass. Thus, even for well-designed components, white box code reuse is occasionally necessary to achieve algorithmic improvements in efficiency.

4 Using Representation Inheritance

In order for `Enhanced_Two_Way_List` to access its superclass' representation safely, we use representation inheritance. With this approach, the author of a subclass such as `Enhanced_Two_Way_List` is *required* to obey the representation invariant(s) and respect the abstraction relation(s) of its superclass(es). In a language that has support for expressing representation invariants and abstraction relations, this requirement could be automatically enforced. Otherwise, it must be enforced by programming conventions and checked through code reviews and testing. Fortunately, well-defined representation invariants and abstraction relations should make the testing of new subclasses much easier—by verifying that subclass methods do in fact respect these superclass assumptions, the need for retesting of inherited methods or other nonlocal code artifacts is greatly reduced.

In the two-way list example, we can change the declaration of the `Two_Way_List` data members from **private** to **protected**, and write down the representation invariant and abstraction relation for its implementation (perhaps in structured comments, since C++ does not support formal descriptions of these assumptions). The author of the `Enhanced_Two_Way_List` class can then have direct access to the representation of list objects when implementing `Swap_Rights`, as long as the invariant and abstraction relation are respected—both must be respected, since `Enhanced_Two_Way_List` is intended to be a behavioral subtype of `Two_Way_List`. Here, "respected" means the following:

> Assume that, before the method is called, the invariant holds on the two-way list object and the abstraction relation gives the correct conceptual value for it. The method then must ensure that upon its completion, the resulting two-way list also satisfies the invariant, and that the abstraction relation gives the correct conceptual value for the new list—one that appropriately reflects the conceptual changes the method was intended to make (i.e., one that conforms to the method's postcondition).

This is the essence of representation inheritance: the flexibility of white box code inheritance is achieved, without giving up the safety afforded by encapsulation of superclass representation information.

The implementation of `Two_Way_List` described in Section 3 relies on several conventions, which taken together form its representation invariant:

1) The `TWL_Nodes` within a `Two_Way_List` object are doubly-connected in a single chain.
2) The `pre_front` and `pre_fence` pointers refer to nodes within the same chain.
3) The unconnected pointers on the sentinel nodes are set to `NULL`.
4) The `pre_front` pointer always refers to the sentinel node at the beginning of the chain.

These conventions are stated informally here, but they could be formalized (with some effort). Some programming languages even provide syntactic slots for expressing representation invariants [4], [3].

In practice, a subclass with white box access to its inherited state variables could fail to maintain any one of these four properties. Fig. 7 gives one example of how an implementation of `Swap_Rights` could violate the first clause of the representation invariant. Fig. 7 depicts the representation of the two lists introduced in Fig. 6 in detail, both before and after the call to `Swap_Rights`. In this case, the implementation of `Swap_Rights` has only exchanged the "next" pointers in the two lists, and has failed to properly switch the corresponding "previous" pointers. Now, neither list's representation is a doubly-connected chain—the two chains are cross-connected. While this can rightly be considered a defect in the implementation of the `Swap_Rights` operation, note that many list operations will continue to operate correctly, and the effects may only be detected by a test suite that exercises the inherited operations interleaved with the new addition in a nontrivial way.

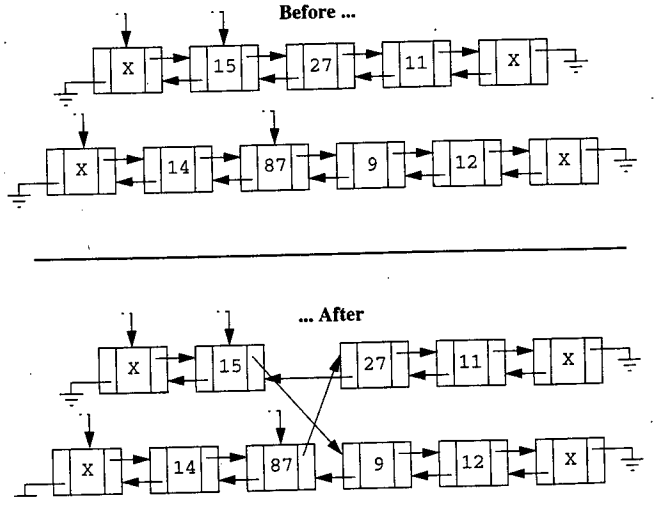


Fig. 7. Violating the representation invariant in `Swap_Rights`.

The abstraction relation then relates representation values to the corresponding conceptual values they realize. It

captures the intentions of the implementer about the "meaning" of the representation—how it encodes the conceptual state that clients reason about. Informally, the doubly-linked chain representation of two-way lists is related to the conceptual model described in Section 3 as follows:

1) The entire sequence of items in the list, as well as their order (i.e., `l.left * l.right`), is recorded by the contents and order of the `TWL_Nodes` in the (single) chain of the representation.
2) The separation between the "left" and "right" parts of the conceptual value (implicitly denoting the "fence") is recorded by the `pre_fence` pointer. Specifically, the `pre_fence` pointer points to the `TWL_Node` containing the *last* item in the "left" portion of the list. The "right" portion of the list begins with the node in the chain immediately following the one pointed to by `pre_fence` (i.e., '`pre_fence->next`').

Some programming languages also provide syntactic slots for expressing abstraction relations or functions [4], [3].

Continuing the running example, Fig. 8 gives one example of how an implementation of `Swap_Rights` could ignore the second clause of the abstraction relation. Here, the implementation of `Swap_Rights` has only exchanged the trailing halves of the two lists, beginning with the nodes pointed to by the "`pre_fence`" pointers. For the two sample lists under consideration, this behavior does not violate the representation invariant and will not cause the execution of any inherited methods to fail at a later point. Instead, there is a mismatch between the behavior described at the conceptual level and the actual representation. When viewed in the light of the abstraction relation described above, it is clear that `Swap_Rights` does not simply exchange everything to the right of the two fences—it also exchanges the item immediately to the left of the fence in each list. Without a description of the abstraction relation, however, the software engineer who wrote this version of `Swap_Rights` might never see the discrepancy.

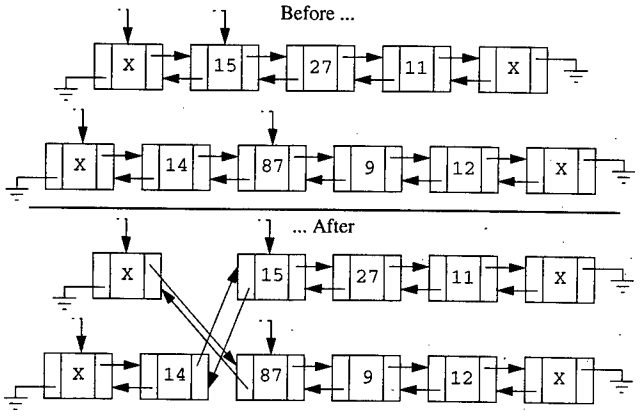


Fig. 8. Violating the abstraction relation in `Swap_Rights`.

The above statements of the representation invariant and the abstraction relation are informal, but they capture critical information necessary for the correct functioning of the `Two_Way_List` methods. There are many other possible configurations of invariant and abstraction relation that could have been chosen (together with slight differences in the choices about the pointers and node structures used). While any of them may work well, the important point is that **one** choice was made in the implementation of the `Two_Way_List` class, and the implementer of that class used it consistently. The correct operation of `Two_Way_List`'s methods critically depends on this choice (and on consistently following it).

Fig. 9 shows an excerpt of the `Two_Way_List` class declaration with an informal version of the representation invariant and abstraction relation added in comments. Fig. 10 then shows the corresponding implementation of `Enhanced_Two_Way_List`'s `Swap_Rights` method, with comments marking the locations where critical assumptions about the inherited representation invariant and abstraction relation must be checked.

```
template <class Item>
class Two_Way_List
{
  // ...
  // The same external decls. As in Figure 3
  // ...
protected:          // Allow representation inheritance
  // The representation of the class:
  struct TWL_Node { ... };
  TWL_Node* pre_front; // ptr. to 1st sentinel
  TWL_Node* pre_fence; // ptr. to node just
                       // before the "fence"
  //! Representation invariant:
  //! Let HEAD_SENTINEL and TAIL_SENT-
  //! INEL be the two sentinel TWL_Nodes for
  //! This list. Then:
  //! 1a. HEAD_SENTINEL.next-> ... -> next
  //!       == &TAIL_SENTINEL and
  //! TAIL_SENTINEL.previous-> ...
  //!           ->previous == &HEAD_SENTINEL
  //! 1b. For all nodes N:
  //!      N.next != NULL =>
  //!        N.next.previous == N and
  //!      N.previous != NULL =>
  //!        N.previous.next == N
  //! 2.  (pre_fence == &HEAD_SENTINEL or
  //!       pre_fence == & HEAD_SENTINEL.next->
  //!       ... ->next) and
  //!     pre-fence != &TAIL-SENTINEL
  //! 3. HEAD_SENTINEL.previous == NULL
  //!    and TAIL=SENTINEL.next ==
  //!    NULL (and nothing else is NULL)
  //! 4. pre_front == &HEAD_SENTINEL
  //! Abstraction relation:
  //! "left"    ==  pre_front-> next-> item,
  //!               pre_front->next->next->item,
  //!               ...
  //!               pre_fence-> item
  //! "right"   ==  pre_frence-> next->item,
  //!               pre_frence->next->item,
  //!               ...
  //!               TAIL_SENTINEL.previous
  //!                   ->item
}
```

Fig. 9. The `Two_Way_List` representation invariant and abstraction relation.

```
template <class Item>
void Enhanced_Two_Way_List<Item>::
    Swap_Rights(Enhanced_Two_Way_List& rhs)
{
  //! assert(Two_Way_List-rep_invariant(self) ==
  //!        true);
  //! Assert(Two_Way_List.abs_relation(
  //!                (self.left, self.right),
  //!                (self.pre_front, self.pre_fence)));

  TWL_Node* my_tail, rhs_tail:

  my_tail = pre_fence→next;
  rhs_tail = rhs.pre_fence→next;

  pre_fence→next = rhs_tail;
  rhs_tail→previous = pre_fence;

  rhs.pre_fence→next = my_tail;
  my_tail→previous = rhs.pre_fence;

  //! assert(Two_Way_List.rep_invariant(self) ==
  //!        true);
  //! assert(Two_Way_List.abs.relation(
  //!                (self.left, self.right),
  //!                (self.pre_front, self.pre_fence)));

  //! My postcondition:
  //! assert ((self-left=#self.left )        &&
  //!         (self.right t=#rhs.right )     &&
  //!         (rhs.left = rhs.left )         &&
  //!         (rhs.right - #self-right));
}
```

Fig. 10. Implementing `Swap_Rights`.

5 Enforcing Obligations

Representation inheritance relies on a subclass living up to the commitments made by its superclass(es). Thus, the safety afforded by representation inheritance is only as strong as the guarantee we have that the subclass will indeed fulfill its obligations. Further, it is clear that this safety is only as strong as the "tightness" of the representation invariants and abstraction functions documented for each class. Failure to capture all the restrictions that superclass methods rely on can still allow too much freedom to subclasses. With regard to gauging safety, there are three basic approaches to establishing the degree to which subclasses obey their representation inheritance restrictions: formal verification, run-time checking, and testing.

5.1 Formal Verification

In theory, formal verification support is needed to provide complete automatic enforcement of representation invariants and abstraction relations. This necessity is brought about by the fact that some representation invariants or abstraction relations may not be computable, implying conformance may not be checkable by a computer through either run-time checks or testing. This should not be surprising, since conformance to a behavioral specification may be just as difficult to check, depending on the specification notation used. Further, regardless of the enforcement technique used, checking adherence to superclass abstraction relations requires that one have behavioral specifications for both super- and subclasses—something lacking in most present software.

In practice, however, few practitioners are willing to proceed with formal verification at present. Instead, code reviews and testing seem to provide acceptably high confidence levels for conformance with behavioral specifications, so we turn our attention to the natural analogues for enforcing representation inheritance restrictions.

5.2 Run-Time Checking

Without the resources for formal verification, many practitioners feel that run-time checking of representation invariants is critical to enforcement. Eiffel is one language which uses run-time checks consistently to provide some level of enforcement for programming obligations [13].

For example, in the `Two_Way_List` component described in Section 3, one could write a protected method that would operationally check the class' representation invariant. This method could then be called directly (perhaps using the standard `assert()` macro) in appropriate places (both in `Two_Way_List` methods and methods of its descendant classes) to ensure that the invariant is being maintained. This would certainly provide some degree of confidence that the necessary obligations imposed on subclasses were being observed.

This is certainly a viable approach to representation inheritance enforcement in many cases. It is important to note that it is not always possible to provide run-time checks (i.e., when the representation invariant is not computable). Further, some run-time checks might be considered prohibitively expensive to consider leaving in place in fielded software. As a result, one might ask the question of whether the same degree of confidence could be obtained without requiring the overhead of run-time checking.

5.3 Enforcement through Testing

The primary strategy for using testing to enforce representation inheritance restrictions is to:

1) Use run-time checking to operationally test representation invariants at all necessary points during testing.
2) Expand white box testing techniques to generate test cases that stress these run-time checks.

This approach utilizes the best features of run-time checking without requiring run-time checks in fielded code. Further, if test case generation takes into account representation invariants, run-time checks during testing may even provide a higher level of confidence the obligations are observed than run-time checks alone.

Simply put, for testing purposes, every class should have a method that operationally checks the representation invariant on its internal state. If all classes export such an operation, it is a simple matter to write "defensive" wrappers for every class that check invariants on entry and exit to every method. By providing such defensive wrappers around superclasses during testing, run-time checking can be systematically inserted where ever it is desired. Generic programming provides and effective way to insert or remove such defensive wrappers without modifying subclasses or their inheritance links [14], [11].

To fully exploit such run-time checks during testing, it is necessary to consider representation invariants when generating test cases. Effectively, the obligation to maintain a

representation invariant becomes an extra part of the behavioral specification of each method (hidden from the eventual clients of a class), and thus is subject to the same test case generation techniques as are used for gauging conventional behavioral conformance.

Once a subclass has been fully tested with run-time checks in place, those checks can be safely removed. Any future subclasses cannot affect code they might inherit, as long as those subclasses obey their representation inheritance obligations. This allows superclasses and sublcasses to be validated independently through testing.

5.4 Testing without Representation Inheritance

In `light` of the previous discussion, it is also worth considering the requirements for testing when the restrictions of representation inheritance are not observed or enforced. Perry and Kaiser [15] describe requirements for adequately testing OO programs. They indicate that when subclasses are added to an inheritance hierarchy, not only must one test the newly added methods in these subclasses, one must also *retest* all of the inherited methods. They also indicate that clients of the superclasses need to be retested while using the subclasses. Component regression testing guidelines built on these requirements have also been described by Skublics et al. [16, p. 85].

To most object-oriented programmers, however, this testing advice is counterintuitive and seems to fly in the face of conventional wisdom. Instead of simply testing the newly added code, one must test all methods in every class. While code reuse may have saved some time during the coding phase of development, according to Perry and Kaiser's recommendations, it saves absolutely no effort in testing. From the testing viewpoint, it is almost as if no inheritance had occurred at all—the testing effort required is the same as if all of the superclass code were reproduced from scratch in the new component.

If one is working in a language where the inheritance mechanism normally allows white box code inheritance, such as Smalltalk or Eiffel, then the technical reasons for Perry and Kaiser's recommendation start to make more sense. When a subclass can cause code outside of itself to fail, testing in the context of newly added subclasses becomes a much more involved process.

From this, we can also infer that white box code inheritance provides little if any savings in maintenance effort. Changing code in one class method could conceivably have adverse affects in arbitrarily distant ancestor or descendant classes. Thus, to make maintenance changes or enhancements in one class, the entire root-to-leaf branch of the inheritance hierachy it lives in must be understood, and then retested after the change—classes fail to provide the fire walls of modularity that programmers expect.

With representation inheritance, the methods inherited from a superclass cannot fail because of defects introduced in subclass method implementations. As a result, changes in a subclass do not require the retesting of inherited code. The virtues of modularity and encapsulation are thus preserved at class boundaries.

6 Relation to Previous Work

As mentioned in the introduction, representation inheritance is related in spirit to various work on specification inheritance. Liskov and Wing's definition of the subtype relation so that it preserves behavioral abstraction typifies this work [17], [1]. In a similar vein, Leavens and Weihl describe a foundation for the modular verification of OO software built around interpreting inheritance as a behavioral abstraction [5]. These approaches only address the client-side reasoning issues posed by inheritance mechanisms, however, and do not directly address code inheritance.

The notions of representation invariant and abstraction relation (or function) [3] are also taken directly from past work on formal specification and formal verification. Both have been used in languages and methods centered on model-based program specification, including RESOLVE [4]. Leavens and Weihl [5] provides a complete formal treatment of representation invariants and abstraction relations in an object-oriented context, and they are naturally extended to other model-based specification approaches, such as those surveyed in Lano and Haughton [18]. This paper gives a more pragmatic presentation in order to familiarize practitioners with the uses of the more theoretical development of representation invariants and abstraction relations presented elsewhere.

The safety problems with white box code reuse have been described by Muralidharan and Weide [19]. They note the efficiency concerns that make white box techniques desirable, but concentrate on clearly delineating the disadvantages that come with breaking encapsulation. They propose no solutions to the problem. The RESOLVE programming language [11] does provide the necessary support for representation invariants and abstraction relations, however, and there are plans to add representation inheritance to the language.

As mentioned in Section 2, Lamping [8] also has examined the risks associated with subclass access during specialization. His work is type-system oriented, however, where the solution proposed here derives from model-theoretic specification techniques. Lamping suggests partitioning classes into groups of methods which share assumptions, as documentation for use by programmers writing specializations. He does not specifically address capturing the assumptions themselves, however.

Stata and Guttag [10] have also explored grouping methods for this purpose. Their work is more closely related, since it is also specification-based. They further propose that instance variables can be partitioned along with the methods, splitting a superclass into "modular" chunks that can be treated independently. While this does allow subfacets of an object to be specialized independently, it fails to capture the critical assumptions about module state upon which the methods depend. Stata and Guttag go on to require that if any method in such a group is overridden by a subclass, then all methods in that group must be, which is necessary for safety. Here, we instead explicitly capture the conventions about how state variables are maintained, we do not require methods to be grouped, and we allow any method to be overridden individually.

Perhaps the best-known work that attempts to address the problems discussed here is Meyer's Eiffel [13]. There are several critical differences between Eiffel and the ideas described in this paper, however, which highlight the contributions of the present paper. At first glance, Eiffel appears to have all of the machinery necessary to capture both specification inheritance and representation inheritance built into the language:

- It supports preconditions and postconditions for describing method behaviors.
- It supports **invariant** assertions to capture properties that methods must preserve when they complete.
- It ensures that each subclass inherits the preconditions, postconditions, and invariants of its superclass(es)—descendants must live up to the obligations of their ancestors.

Unfortunately, under practical usage these mechanisms are not enough to ensure the safety that representation inheritance provides.

Classes in Eiffel represent component *implementations*, and there is no linguistic facility for capturing the corresponding component specifications [13, p. 59]. As a result, the mechanisms in the language only support capturing information relevant to the implementation, and other details such as abstraction relations are not addressed.

As a result, Eiffel's **invariant** assertions must serve double-duty:

1) Programmers try to use them to capture the *abstract* invariant [3, p. 92], which defines client-visible constraints on an object's conceptual value.
2) They should also capture the *representation* invariant, which defines constraints on an object's internal state that is invisible to clients.

Of course, assertions that deal with the hidden state of objects are not helpful for client understanding, so it is common to see Eiffel invariants phrased in terms of publicly visible accessor functions [12] rather than private state variables. As a result, Eiffel's **invariant** assertions become abstract invariants in practice.

This tendency is exemplified by Meyer's version of ***TWO_WAY_LIST*** [12, pp. 154–155, 299–303]. The Eiffel version has its invariant phrased in terms of publicly visible accessors, and simply constrains client-visible properties, like the relationships between the number of items in the list, the position of the fence, and the values of predicates similar to `At_Start()` and `At_Finish()`. *None* of the representation-level constraints shown in Fig. 9 are captured.

In addition, the computational nature of Eiffel's assertion mechanism prevents some invariants from being expressed because they are not computable, and discourages programmers from writing down others that are expensive to check. For example, consider a component that implements an associative mapping using a hash table with sorted buckets. The fact that the buckets are maintained in sorted order, and that every key in the mapping is unique, are invariant properties of this implementation. Unfortunately, it is expensive to check these properties at run-time, perhaps prohibitively so. As a result, facets of the component's representation invariant may be ignored by component designers when writing Eiffel assertions.

Finally, the lack of separate specifications in Eiffel ensures that abstraction relations will not be captured. In the `Two_Way_List` example, the assumption that `pre_fence` points to the node *before* the first item in the right half of the conceptual value of the list cannot be captured in an Eiffel **invariant** clause. As a result, subclass methods could violate this assumption, perhaps by leaving a particular list so that the `pre_fence` pointed to the node holding the first item in the right half of the list. This error could potentially cause other methods to fail, or simply have the incorrect behavioral result from the client's point of view. Either way, however, Eiffel assertions cannot address the issue.

While Eiffel's inheritance rules attempt to achieve the same goal as representation inheritance in spirit, in practice none of the assumptions recorded for the `Two_Way_List` example in Fig. 9 would have been captured or checked in a typical Eiffel version of the component. Indeed, none are for the most similar components in Meyer's library. Eiffel fails to provide the safety of representation inheritance for this reason.

7 Conclusions

Class designers have a choice between black box and white box techniques when they specialize existing classes. While it is always best in principle to use black box code inheritance, there are practical situations where programmers really desire more freedom of access to information encapsulated within superclasses. When these situations arise, white box code inheritance is appropriate.

Unrestricted white box code inheritance is clearly unsafe, however. By breaking the encapsulation of superclasses, it allows subclass implementers to violate assumptions upon which superclass methods depend. This can mean that subclasses actually introduce errors that are only observed through execution of inherited methods, making it impossible to reason about class correctness locally, and seriously complicating the requirements for adequate testing of software.

If the assumptions that classes depend on are described in terms of representation invariants and abstraction relations, then it is possible to address the shortcomings of white box reuse. Representation inheritance is a controlled form of white box code inheritance in which subclasses must respect the representation assumptions of their ancestors. By doing so, subclasses ensure that superclass code assumptions are protected, while simultaneously enjoying the benefits of direct access to superclass state representations. This gives desirable freedom to subclass implementers, while preserving the safety and locality considerations for which all programmers strive.

Acknowledgments

The author gratefully acknowledges financial support from the National Science Foundation under Grant No. CCR-9311702 and the Advanced Research Projects Agency under Contract No. F30602-93-C-0243 (monitored by the USAF Materiel Command, Rome Laboratories, ARPA order number A714). Bruce Weide also deserves special thanks for

suggesting the ideas that later developed into the notion of representation inheritance presented here.

REFERENCES

[1] B.H. Liskow and J.M. Wing, "A Behavioral Notion of Subtyping," *ACM Trans. Programming Languages and Systems*, vol. 16, pp. 1,811–1,841, Nov. 1994.

[2] S.H. Edwards, "An Approach for Constructing Reusable Software Components in Ada," IDA Paper P-2378, Inst. For Defense Analyses, Alexandria, Virginia, Sept. 1990.

[3] B. Liskov and J. Guttag, "Abstraction and Specification in Program Development," *The MIT Electrical Engineering and Computer Science Series*, Cambridge, Mass.: MIT Press, 1986.

[4] P. Bucci, J.E. Hollingsworth, J. Krone, and B.W. Weide, "Implementing Components in RESOLVE," *ACM SIGSOFT Software Eng. Notes*, vol. 19, pp. 50–52, Oct. 1994.

[5] G.T. Leavens and W.E. Weihl, "Specification and Verification of Object-Oriented Programs Using Supertype Abstraction," *Acta Informatica*, vol. 32, no. 8, pp. 705–778, 1995.

[6] J.M. Wing, "A Specifier's Introduction to Formal Methods," *Computer*, vol. 23, no. pp. 8–24, Sept. 1990.

[7] M. Sitaraman, L.R. Welch, and D.E. Harms, "On Specification of Reusable Software Components," *Int'l J. Software Eng. and Knowledge Eng.*, vol. 3, no. 2, pp. 207–229, 1993.

[8] J. Lamping, "Typing the Specialization Interface," *Proc. Conf. OOPSLA'93*, pp. 201–214, *ACM*, Oct. 1993.

[9] J. Lamping and M. Abadi, "Methods as Assertions," *ECOOP'94—Object-Oriented Programming, Proc. Eighth European Conf.*, Lectures Notes in Computer Science, vol. 821, pp. 60–80. New York: Springer-Verlag, 1994.

[10] R. Stata and J.V. Guttag, "Modular Reasoning in the Presence of Subclassing," *Proc. Conf. OOPSLA'95*, pp. 200–213. New York: ACM, 1995.

[11] M. Sitaraman and B.W. Weide, eds., "Special Feature: Component-Based Software Using RESOLVE," *ACM SIGSOFT Software Eng. Notes*, vol. 19, pp. 21–67, Oct. 1994.

[12] B. Meyer, *Reusable Software: The Base Object-Oriented Component Libraries*. Hertfordshire, UK: Prentice Hall Int'l, 1994.

[13] B. Meyer, *Object-Oriented Software Construction*. New York: Prentice Hall, 1988.

[14] J. Hollingsworth, "Software Component Design-for-Reuse: A Language Independent Discipline Applied to Ada," PhD thesis, Dept. of Computer and Information Science, Ohio State Univ., Columbus, 1992.

[15] D.E. Perry and G.E. Kaiser, "Adequate Testing and Object-Oriented Programming," *J. Object-Oriented Programming*, vol. 2, pp. 13–19, Jan./Feb. 1990.

[16] S. Skublics, E.J. Klimas, and D.A. Thomas, *Smalltalk with Style*. Englewood Cliffs, N.J.: Prentice Hall, 1996.

[17] B.H. Liskov and J.M. Wing, "A New Definition of the Subtype Relation," *ECOOP'93—Object-Oriented Programming, Proc. Seventh European Conf*, Lecture Notes in Computer Science, vol. 707, pp. 118–141, New York: Springer-Verlag, 1993.

[18] K. Lano and H. Haughton, eds., *Object-Oriented Specification Case Studies*. Englewood Cliffs, N.J.: Prentice Hall, 1993.

[19] S. Muralidharan and B.W. Weide, "Should Data Abstraction be Violated to Enhance Software Resuse?" *Proc. Eighth Ann. Nat'l Conf. Ada Technology*, Atlanta, pp. 515–524, ANCOST, Inc., Mar. 1990.

Stephen H. Edwards received the BSEE degree from the California Institute of Technology, Pasadena, and the MS and PhD degrees in computer and information science from the Ohio State University, Columbus. He is currently a visiting assistant professor in the Department of Computer Science at the Virginia Polytechnic Institute and State University. He also maintains a close relationship with the Reusable Software Research Group at the Ohio State University. His research interests include software engineering and reuse, the use of formal methods in programming languages, and information retrieval technology. Dr. Edwards is a member of the IEEE Computer Society, ACM, and Upsilon Pi Epsilon.

On the Practical Need for Abstraction Relations to Verify Abstract Data Type Representations

Murali Sitaraman, *Member, IEEE Computer Society*, Bruce W. Weide, *Member, IEEE*, and William F. Ogden, *Member, IEEE Computer Society*

Abstract—The typical correspondence between a concrete representation and an abstract conceptual value of an abstract data type (ADT) variable (object) is a many-to-one function. For example, many different pointer aggregates give rise to exactly the same binary tree. The theoretical possibility that this correspondence generally should be relational has long been recognized. By using a nontrivial ADT for handling an optimization problem, we show why the need for generalizing from functions to relations arises naturally in practice. Making this generalization is among the steps essential for enhancing the practical applicability of formal reasoning methods to industrial-strength software systems.

Index Terms—Abstract data type, abstraction function, abstraction mapping, abstraction relation, data abstraction, formal specification, greedy algorithm, program verification, nondeterminism, optimization problem, relation.

1 Introduction

THE need to separate the specifications and implementations of abstract data types is widely recognized. To keep a specification purely conceptual and unbiased with respect to its many alternative implementations, the behavioral explanation should employ an implementation-neutral abstract model rather than any particular representation model. The formal verification that a given implementation does meet this conceptual specification then involves a correspondence mapping, traditionally called an *abstraction function*, between the model used in the implementation (the concrete or representation model) and the model used in the specification (the abstract or conceptual model) [10].

For some ADT specifications and implementations, the natural connection between concrete and abstract models turns out to be relational, not functional. That is, in some cases a particular concrete value may represent any of several abstract values; see Fig. 1.

The theoretical importance of abstraction relations has long been recognized. Precluding their expression results in modular verification systems which are incomplete in the technical sense that there are implementations that are correct with respect to their specifications, but which cannot be proved to be so using only abstraction functions. Moreover, insisting upon using an abstraction function even when it is technically possible may increase verification complexity to the point where it effectively thwarts modular reasoning about correctness. And it is crucial for tractability and reuse that the verification of an ADT's implementation code should be modular. This means that the proof of correctness should rely only on the given specification of behavior to be implemented and on given specifications of lower-level components that are used in the code. The correctness argument should be independent of the implementations of the lower-level components and independent of other parts of the system that use the code being verified [7], [23].

Here, we formally establish the requirement for supporting abstraction relations by exhibiting a nontrivial ADT for a practical optimization problem, where not just the value of—but the outright need for—an abstraction relation naturally arises. The nature of the example argues that formal reasoning systems must be able to generalize to handle abstraction relations if they are to be applied with confidence to new and nontrivial data abstractions.

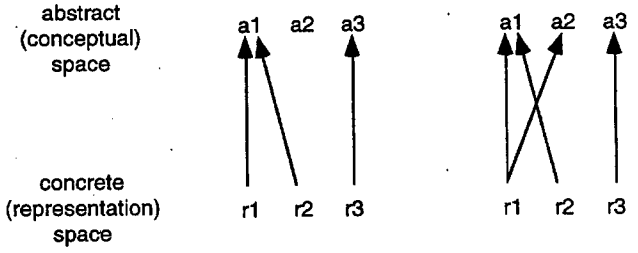


Fig. 1. Abstraction function (left) and abstraction relation (right).

1.1 Prior Work on Abstraction Relations

Previous work involving modular verification of ADTs with model-based specifications leaves the practical role of abstraction relations unsettled. Leavens notes the value of "simulation relations" (essentially abstraction relations) in defining behavioral subtyping [13]. Jones [11, p. 219] and Schoett [19] independently observe that, technically, ab-

• *M. Sitaraman is with the Department of Statistics and Computer Science, West Virginia University, Morgantown, WV 26506. E-mail: murali@cs.wvu.edu.*
• *B.W. Weide and W.F. Ogden are with the Department of Computer and Information Science, The Ohio State University, Columbus, OH 43210. E-mail: {weide, ogden}@cis.ohio-state.edu.*

Manuscript received Sept. 18, 1995; revised Feb. 14, 1997.
Recommended for acceptance by J.D. Gannon.
For information on obtaining reprints of this article, please send e-mail to: transse@computer.org, and reference IEEECS Log Number S95823.

straction relations might be needed in some cases to verify implementations of ADTs whose specifications are "biased," or "not fully abstract". Schoett's work is based on the assumption that nonfully abstract specifications might arise in practice. But Jones notes that [11, p. 182], "If different abstract values correspond to one concrete value, it is intuitively obvious that such values could have been merged in the abstraction. So, in the situation where the objects used in the specification were abstract enough, the many-to-one situation would not arise."

There also has been some work on abstraction relations in the context of algebraic specifications. For example, to show the theoretical need for abstraction relations, Nipkow describes a construction involving algebras of nondeterministic data types [18]. The relationship between this work and the practical need for abstraction relations to verify implementations of model-based specifications is unclear, however. So while there are subtle differences in the positions taken by different authors on the topic, the comments of Liskov and Wing in their corrigendum to an earlier paper [14] probably best characterize the common belief among software engineers who use formal methods: Abstraction relations are occasionally helpful and might even be technically necessary in some cases; however, [15, p. 4] "for most practical purposes, abstraction functions are adequate (compared to relations)."

1.2 Contributions

We show that abstraction relations are practically important for software specification and modular verification. Technically, abstraction relations are necessary in order to avoid incompleteness. Practically, they are necessary in order to deal with new and nontrivial ADTs such as those resulting from modern software component design techniques.

We use a sample specification based on the technique of "recasting" algorithms as data abstractions [24]. This software component, if it is to lend support to the claim for practical significance of abstraction relations, must have three properties:

1) *Realism.* The specification must not be artificial and conceived just for showing the need, i.e., it must be of a sort that is actually likely to arise in practical systems. Otherwise, the fact that a reasoning system based solely on abstraction functions cannot handle it would have little practical import. Our sample specification captures solutions to a practical optimization problem and serves as an exemplar for a larger class of similar components.
2) *Quality.* The sample specification must be well-designed. In particular, it must be fully abstract; i.e., every two different conceptual values of the abstract data type being defined must be computationally distinguishable [11], [12], [25]. Otherwise, the relational nature of the correspondence mapping could merely arise from the sloppiness of the conceptual specification. Our sample component is a well-designed, fully abstract specification.
3) *Provable resistance to verification with abstraction functions.* There must be an actual proof that shows why no abstraction function can be found to verify that a correct implementation satisfies the sample specification. Our sample component comes with a correct and practical realization that we prove cannot be verified using any abstraction function (but which can be verified using an abstraction relation).

2 Inherently Relational Behavior Specifications

Optimization problems are a general category of problems in which relational specifications arise naturally. In many such problems, it is easy to find multiple solutions which satisfy the constraints yet which all evaluate to the same objective function value. The specification for software to solve such a problem is inherently relational because it should allow an implementation to produce *any* optimum solution. The natural correspondence between such implementations and specifications tends to be relational (even though a functional correspondence might exist in some cases).

2.1 A Realistic Software Component Example

As a sample relational problem specification we use the `Spanning_Forest_Machine_Template` from our recent paper on "recasting" algorithms as objects [24]. This specification exhibits the relational behavior we seek because it requires that *some* minimum spanning forest (MSF) of a given graph must be found; there might be ties and any best answer is acceptable. For a fully connected graph an MSF is also a minimum spanning tree (MST). For a general unconnected graph, an MSF is a union of edges of MSTs for each of the connected components [4].

The concept for `Spanning_Forest_Machine_Template` defines a type `Spanning_Forest_Machine` (a variable of which type we henceforth call a "machine" for brevity) and suitable operations. A typical client repeatedly calls operation `Insert` to add the edges of the graph for which an MSF is to be found (one at a time) into a machine; calls `Change_To_Extraction_Phase` to change the machine to extraction phase, and finally makes multiple calls to `Extract` to remove, one at a time, the edges of one of the (possibly many) MSFs of that graph. Operation `Insert` requires that the machine be in the insertion phase at the time of the call, whereas `Change_To_Extraction_Phase` and `Extract` operations require that the machine be in the extraction phase. `Is_In_Insertion_Phase` tests whether a machine is in insertion phase. `Size` returns the number of MSF edges in the graph and is restricted to be called only when in the extraction phase (for purposes of simplicity in this paper).

The concept described informally above, and specified formally in Fig. 2, is quite different from one providing a single procedure that finds an MSF of a graph. Our component prescribes what computation needs to take place, but *not when*. Viewed through its abstract interface, the component does not reveal to a user when (i.e., in which operation or operations) an MSF is actually being computed. The design gives the implementer freedom both in how and in when to do computations, and the attendant performance flexibility of various kinds of cost amortization, which is part of the rationale for the recasting technique illustrated by this interface [24]. This observation reinforces an important principle

```
concept Spanning_Forest_Machine_Template

   context
      global context
         facility Standard_Boolean_Facility
         facility Standard_Integer_Facility

      parametric context
         constant max_vertex: Integer
            restriction max_vertex > 0

      local context
         math subtype EDGE is (
                  v1: integer
                  v2: integer
                  w: integer
               )
            exemplar e
            constraint
               1 <= e.v1 <= max_vertex and
               1 <= e.v2 <= max_vertex and
               e.w > 0
         math subtype GRAPH is finite set of EDGE
         math operation IS_MSF (
               msf: GRAPH
               g: GRAPH
            ): boolean
            definition
              (* true iff msf is an MSF of g *)

   interface
      type Spanning_Forest_Machine is modeled by (
                edges: GRAPH
                insertion_phase: boolean
             )
         exemplar m
         constraint IS_MSF (m.edges, m.edges)
         initialization ensures
            m = (empty_set, true)

      operation Change_To_Insertion_Phase (
               alters      m: Spanning_Forest_Machine
            )
         requires
            not m.insertion_phase
         ensures
            m = (empty_set, true)

      operation Insert (
              alters      m: Spanning_Forest_Machine
              consumes    v1: Integer
              consumes    v2: Integer
              consumes    w: Integer
            )
         requires
            m.insertion_phase and
            1 <= v1 <= max_vertex and
            1 <= v2 <= max_vertex and
            w > 0
         ensures
            IS_MSF (m.edges, #m.edges union {(#v1, #v2, #w)}) and
            m.insertion_phase

      operation Change_To_Extraction_Phase (
              alters      m: Spanning_Forest_Machine
           )
         requires
            m.insertion_phase
```

```
    ensures
      m = (#m.edges, false)

  operation Extract (
        alters      m: Spanning_Forest_Machine
        produces    v1: Integer
        produces    v2: Integer
        produces    w: Integer
      )
    requires
      m.edges /= empty_set and
      not m.insertion_phase
    ensures
      (v1, v2, w) is in #m.edges and
      m = (#m.edges - {(v1, v2, w)}, false)

  operation Size (
        preserves  m: Spanning_Forest_Machine
      ): Integer
    requires
      not m.insertion_phase
    ensures
      Size = |m.edges|

  operation Is_In_Insertion_Phase (
        preserves  m: Spanning_Forest_Machine
      ): Boolean
    ensures
      Is_In_Insertion_Phase = m.insertion_phase

end Spanning_Forest_Machine_Template
```

Fig. 2. Specification of `Spanning_Forest_Machine_Template`.

for implementers of model-based specifications: They must always distinguish abstract models from concrete representations and must not let the abstract view bias how or when to manipulate the concrete representation.

2.2 Recasting and Abstraction Relations

We might have used any of a number of recasting examples in this paper. When optimization algorithms, such as those for finding MSFs as well as others such as those for finding single-source shortest paths, are recast as data abstractions, abstraction relations arise naturally in verifying some of their implementations. To see this general need for an entire class of situations similar to the one used in our sample, consider the relational specification of any graph optimization problem where the output is specified to be any one of many possible optimum values. Assume that the specification delineates two distinct phases as in the case of `Spanning_Forest_Machine_Template`: an insertion phase in which edges of a graph can be inserted and an extraction phase in which an optimum answer (say, a set of edges) can be extracted one at a time.

A straightforward model of the ADT defined by the above specification might be an ordered triple: a boolean *phase* that indicates the phase of machine *m*, an *input* set of edges that captures the graph edges inserted into *m*, and an *output* set of edges that defines an optimum solution. Initially, *phase* indicates insertion phase, and *input* and *output* are empty sets. The specification of the `Insert` operation changes only *input* as it adds a new edge. The postcondition of `Change_To_Extraction_Phase` is relational and dictates merely that *output* should become an optimum solution for *input*. The `Extract` operation is specified to return one of the remaining edges of *output*. In this specification, then, it appears that a solution is computed in "batch" fashion when `Change_To_Extraction_Phase` is called.

But other implementations might be possible and reasonable. Consider an amortized cost implementation that accumulates graph edges during the insertion phase but does no special computation in the `Insert` or `Change_To_Extraction_Phase` operations; it computes and returns each edge of an optimum solution only incrementally whenever an `Extract` operation is called. For example, this is how any "greedy" algorithm might be naturally amortized. In the extraction phase, the natural correspondence between the internal representation and the abstract model is relational. It is of the general form:

IS_AN_OPTIMUM_SOLUTION (m.output, S(m.rep))

where *S* is a function from the specific representation of *m* to the mathematical set of edges not yet processed. While there might exist abstraction functions for some implementations such as outlined here, since abstraction relations introduce no significant additional complexity to verification and may actually simplify the conditions—as argued later in this paper—a practical formal system should facilitate the use of abstraction relations in cases like this where they are natural.

The `Spanning_Forest_Machine_Template` can be specified in ways other than the one outlined above [22]. But regardless of how the concept is specified, abstrac-

tion relations arise because the specification needs to capture and allow only MSFs of the input graphs, whereas some implementations might not compute an MSF when the specification suggests. Such situations are typical when the recasting technique is employed.

3 The Practical Need for Abstraction Relations

Section 2 addressed the realistic nature of our sample component. Now we introduce its formal specification; demonstrate its quality in the sense that this specification is fully abstract and, therefore, not defective in the sense discussed in Section 1.1 [11]; and finally prove that there are practical and correct implementations of this component that cannot be verified using any abstraction function.

3.1 A Formal Specification of `Spanning_Forest_Machine_Template`

Fig. 2 is a reproduction of the `Spanning_Forest_Machine_Template` specification from [24] as expressed in the model-based specification language RESOLVE [21].[1] The specification language does not affect the issues discussed in this paper. Any model-based formal specification language [26] would suffice.

To specify the behavior of the operations described informally in Section 2, we model a value of type `Spanning_Forest_Machine` as an ordered pair: a weighted graph *edges* which is treated as a finite set of positively-weighted edges, and a boolean flag *insertion_phase* which is true iff the machine is in the insertion phase. The specification defines and uses a mathematical predicate *IS_MSF(msf, g)* which is true iff the graph *msf* is a minimum spanning forest of the graph *g*. The details of this definition are elided in Fig. 2 but they are straightforward.

From the specification it *appears* that a machine in insertion phase retains only an MSF for the edges inserted so far—not the entire set of edges inserted so far—thus giving an external observer the impression that an MSF is kept incrementally all along. But, as noted earlier, because a client of the component cannot see the representation, an implementation actually might keep all the inserted edges until `Change_To_Extraction_Phase` is called and then batch-process them to weed out nonMSF edges; or it might use an amortized cost implementation.

The specification in Fig. 2 raises an important question: Does it really allow an implementation to produce during the extraction phase *any* MSF of the inserted edges, or does the specified incremental nature of the `Insert` operation rule out some possible MSFs? It turns out that the specification is not restrictive, a fact that follows directly from a lemma from graph theory about MSF properties:

$$\forall G_1, G_2 : GRAPH,\ e : EDGE$$
$$(IS\_MSF(G_1, G_2 \cup \{e\}) \Rightarrow$$
$$\exists G_3 : GRAPH(IS\_MSF(G_3, G_2) \wedge IS\_MSF(G_1, G_3 \cup \{e\})))$$

1. A summary of RESOLVE specificaiton notations essential for understanding this paper is given in Appendix A. There are a few minor changes in this specification from the one in [24] to reflect current RESOLVE syntax. The one substantive change is that the `Size` operation here has a precondition; it cannot be called during the insertion phase (but there is no reason to do so in any case). This change permits a simplified presentation in Section 3 but does not materially affect any of the issues we raise.

A proof of the lemma involves standard arguments from graph theory, where a case analysis based on whether *e* is in G_1 yields a construction for G_3. The proof of correctness of a batch-style implementation of `Spanning_Forest_Machine_Template` (see Appendix B) explains the relevance of this lemma and the conclusion that it demonstrates why the specification in Fig. 2 is not restrictive.

We conclude this section by noting that the specification in Fig. 2 is fully abstract, i.e., it is both "observable" and "controllable" [25]. To be observable (hence fully abstract), the specification must make it possible to distinguish every two abstract model values through the provided operations [11], [12]. We specify that a machine keeps only MSF edges at all times; clearly any two different MSFs can be distinguished by a sequence of calls to `Extract` operations. To be controllable, the specification must make it possible to construct every abstract model value. This also is permitted because any particular MSF can be constructed through an appropriate sequence of calls to `Insert` with just that MSF's edges.

3.2 A Class of Implementations that Need Abstraction Relations

To prove the practical need for abstraction relations, it remains to show the existence of a valid and practical implementation of this specification that cannot be proved correct using any abstraction function, but which can be verified using an abstraction relation. The argument is organized as follows. First we characterize a set of valid "nonmonotonic, deterministic, batch-style" implementations, any of whose members could serve as this unverifiable implementation. Next we show why these implementations cannot be proved correct using *any* abstraction function. In the last subsection, we explain how abstraction relations can be used to verify these implementations in a modular proof system [7].

In this discussion, it is important to note that the specificity of the particular class of implementations to be considered arises only because we seek to show the resistance to verification using abstraction functions, with minimal "hand waving." Other than this there is nothing special about the class of implementations considered. Amortized cost implementations, for example, would have served equally well.

3.2.1 Deterministic Batch-Style Implementations

Let ***B*** be the class of valid deterministic batch-style implementations of `Spanning_Forest_Machine_Template`. The implementations in ***B*** are, first, *deterministic*: the outputs computed by each operation are entirely determined by its inputs. Two abstract operations (`Insert` and `Extract`) have behavioral specifications that are relational, but their implementations may have deterministic functional behavior. In order to be valid, an implementation need only exhibit a behavior pattern that is consistent with the specified relation; it is not necessary for the implementation actually or even potentially to give different results when run multiple times with the same inputs. We restrict our attention to deterministic implementations both because deterministic behavior for an

implementation is a typical situation in practice, and because this determinism simplifies the proof that abstraction relations are required for verification.

The implementations in B also are *batch-style*. This means they just store all the inserted graph edges while a machine is in insertion phase, deferring computation of a minimum spanning forest to the start of the extraction phase. So initially, the edge collection representing a machine state is empty.[2] The `Insert` operation adds a new edge to it. `Change_To_Extraction_Phase` computes a minimum spanning forest of the edge collection (e.g., using Kruskal's algorithm) and stores only the resulting MSF edges back into the edge collection. The `Extract` operation simply removes and returns any edge from the edge collection. The `Size` operation returns the number of edges in that collection, and `Change_To_Insertion_Phase` empties it.

Why are such batch-style implementations correct, i.e., why should we consider them to behave as specified? This question about correctness has to do with the timing of events: Is it possible for a client of `Spanning_Forest_Machine_Template` to detect that a batch-style implementation is being used, as opposed to an "eager beaver" implementation that seems natural from the specification? The behavior of any implementation of any abstraction can be detected only through the "observer" operations provided in its interface, in this case `Extract` and `Size`. It is clear that the `Size` operation as described above produces the specified result because its precondition limits it to being called during extraction phase, where the representation used in a batch-style implementation contains precisely the same edges as in the conceptual view. `Extract` as described above also works as advertised because, before it can be called, `Change_To_Extraction_Phase` has computed an MSF of the graph that was input during the insertion phase. The apparent discrepancy between the specification and a batch-style implementation with respect to *when* computation of an MSF occurs (seemingly incrementally during the insertion phase according to the specification, but actually in `Change_To_Extraction_Phase` in the implementation) simply cannot be detected by a client from functional behavior alone. So a batch-style implementation is as good as any other from this perspective.

3.2.2 Monotonic Deterministic Batch-Style Implementations

A deterministic batch-style implementation I that can be verified with an abstraction function exhibits an interesting property we term *monotonicity*, denoted *Mono*(I). Consider the client code labeled `Find_MSF` which takes, as input, a sequence of edges $E_n = \langle e_1, e_2, \ldots, e_n \rangle$ and produces as output a set of edges $F_n = \{f_1, f_2, \ldots, f_k\}$. (The output order is irrelevant, so we view the output edges simply as a set, not a sequence.)

```
Find_MSF:
   if not Is_In_Insertion_Phase (m) then
      Change_To_Insertion_Phase (m)
   end if
   for i in 1..n loop
      let (v1, v2, w) = e_i
      Insert (m, v1, v2, w)
   end loop
```

```
Change_To_Extraction_Phase (m)
k = Size (m)
for i in 1..k loop
   Extract (m, v1, v2, w)
   let f_i = (v1, v2, w)
end loop
```

Given any $I \in B$ as the underlying implementation of `Spanning_Forest_Machine_Template`, suppose we run `Find_MSF` on $E_n = \langle e_1, e_2, \ldots, e_n \rangle$, producing output F_n; and we run it on $E_{n+1} = \langle e_1, e_2, \ldots, e_n, e_{n+1} \rangle$, producing output F_{n+1}. Then we define:

$$Mono(I) \Leftrightarrow \forall E_{n+1}(IS\_MSF(F_{n+1}, F_n \cup \{e_{n+1}\}))$$

That is, a deterministic batch-style implementation I is monotonic iff the output of `Find_MSF` using I, on any extension of any original input sequence E_n, is an MSF of the same extension of the original sequence's MSF F_n.

Using this property, we define the set of monotonic batch-style implementations:

$$M = \{I \mid I \in B \wedge Mono(I)\}$$

3.2.3 Sample Execution of a Nonmonotonic Deterministic Batch-Style Implementation

In Section 3.3, we will see that deterministic batch-style implementations which can be verified using abstraction functions are monotonic, so the implementations in $B - M$ interest us. The obvious question is whether there *are* any such implementations and, if so, whether they have practical significance. Fig. 3 helps us answer both questions affirmatively by showing the behavior of `Find_MSF` on an example for an implementation $I \in B - M$. In the figure, heavy lines depict possible minimum spanning forests of graphs; thin lines depict other edges inserted so far ("redundant" edges); and program states are identified by letters on the left.

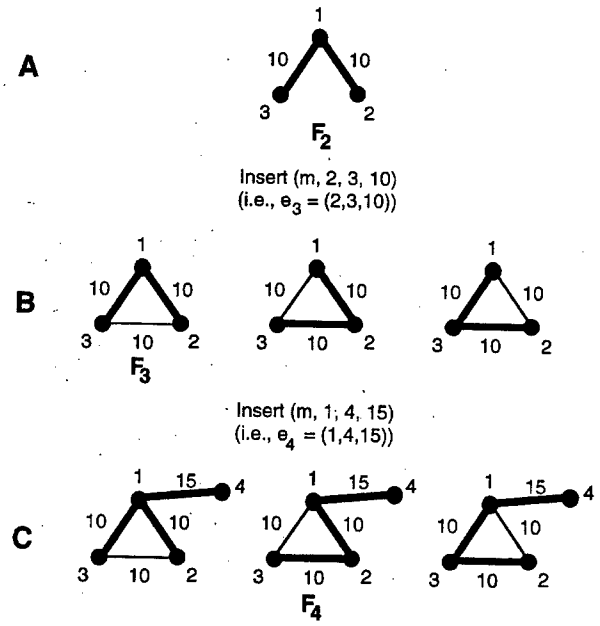


Fig. 3. Sample execution of `Find_MSF` for an implementation $I \in B - M$.

2. The representation also includes a flag indicating the machine's phase, which is handled in the obvious way and therefore is not discussed further.

Starting with an empty machine m in insertion phase, we first insert edges (1, 2, 10) and (1, 3, 10), in either order (i.e., E_2 = <(1, 2, 10), (1, 3, 10)> or E_2 = <(1, 3, 10), (1, 2, 10)>). At this point, state A, there is only one possible minimum spanning forest of the edges inserted so far. Calling `Change_To_Extraction_Phase` and then `Extract` until the machine is empty produces F_2 = {(1, 2, 10), (1, 3, 10)}. That is, `Find_MSF` on input E_2 outputs F_2.

Instead of calling `Change_To_Extraction_Phase` after state A, suppose we insert edge e_3 = (2, 3, 10). If the insertion phase ends at state B, `Find_MSF` returns one of three possible MSFs. Suppose (without loss of generality) it is the leftmost one in state B, so `Find_MSF` on input E_3 produces F_3 = {(1, 2, 10), (1, 3, 10)}. The input sequence so far is *not* a witness to the nonmonotonicity of I because F_3 *is* an MSF of $F_2 \cup \{e_3\}$.

However, suppose we continue in state B to insert one more edge e_4 = (1, 4, 15). If the insertion phase ends at state C, `Find_MSF` again returns one of three possible MSFs. But now suppose it is the middle one in state C, so `Find_MSF` on input E_4 produces F_4 = {(1, 2, 10), (2, 3, 10), (1, 4, 15)}. This input sequence *is* a witness to the nonmonotonicity of I because F_4 is *not* an MSF of $F_3 \cup \{e_4\}$. (In the figure, note that F_3 and F_4 include only the heavy edges.)

Are there really valid batch-style implementations of `Spanning_Forest_Machine_Template` that behave as in Fig. 3? While the answer to this question would be true even if there were only pathological programs that behave this way, the notable feature of the present example is that there is a large and natural class of implementations that serve as exemplars. For instance, implementations that do not keep the edges in input order during the insertion phase, and those that use typical published code for Kruskal's algorithm [4] and are based on sorting algorithms which are not necessarily stable (e.g., quicksort or heapsort), are all examples of actual implementations in $\boldsymbol{B - M}$.

3.3 Inadequacy of Abstraction Functions

We wish to prove that if $I \in \boldsymbol{B}$ (call this proposition p) and $I \notin \boldsymbol{M}$ (proposition q), then I cannot be verified using an abstraction function (proposition r). Notice that:

$$\begin{aligned}(p \wedge q) \Rightarrow r &\equiv \neg r \Rightarrow \neg(p \wedge q)\\ &\equiv \neg r \Rightarrow (\neg p \wedge \neg q)\\ &\equiv (p \wedge \neg r) \Rightarrow \neg q\end{aligned}$$

So, we begin by assuming $I \in \boldsymbol{B}$ (i.e., proposition p) and that I can be verified using an abstraction function (i.e., $\neg r$). We show this implies $I \in \boldsymbol{M}$ (i.e., $\neg q$).

Let A be the abstraction function used in the assumed proof of I. It maps a representation of a `Spanning_Forest_Machine` (call it *m.rep*) to the corresponding conceptual value *m.edges*.[3] Now observe the detailed operation of `Find_MSF` by considering the trajectory of *m.rep* as `Find_MSF` executes with arbitrary input sequence E_n = <e_1, e_2, ..., e_n>, as illustrated in Fig. 4 (top trajectory). There, $m.rep_i$ denotes the representation state immediately after the call that inserts e_i into m; $m.rep_{i'}$ the representation state immediately after the call that extracts f_i from m; and $m.rep_0$ ($m.rep_{0'}$) the state immediately before the first `Insert` (`Extract`) operation.

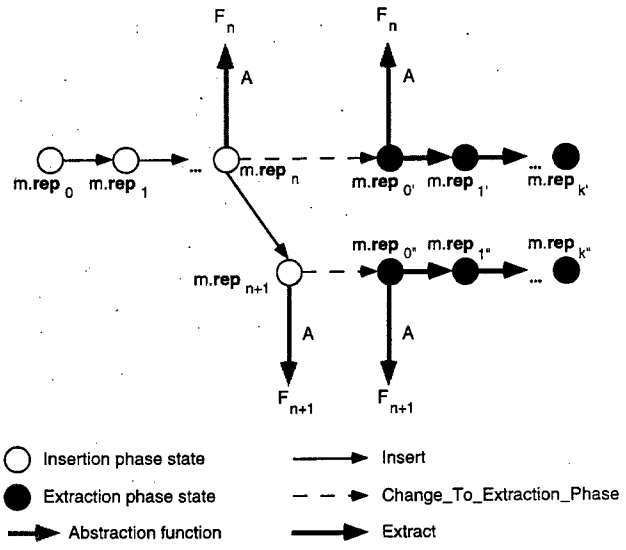


Fig. 4. Behavior of `Find_MSF` for E_{n-1} and E_n.

After all edges are inserted, there is a call to `Change_To_Extraction_Phase`. Because I is a batch-style implementation we expect that $m.rep_{0'} \neq m.rep_n$. However, by the assumption that I can be verified we know from the postcondition of `Change_To_Extraction_Phase` that *m.edges* does not change; thus:

$$A(m.\boldsymbol{rep}_{0'}) = A(m.\boldsymbol{rep}_n) \tag{1a}$$

By the same assumption, we know that each subsequent call to `Extract` removes one edge from $A(m.rep_{0'})$. This means the loop in `Find_MSF` produces F_n as its output; so:

$$F_n = A(m.\boldsymbol{rep}_{0'}) \tag{1b}$$

The trajectory of *m.rep* as `Find_MSF` executes with input sequence E_{n+1} = <e_1, e_2, ..., e_n, e_{n+1}> is similar. Because I is deterministic, the representation state follows precisely the same trajectory as before through insertion of edge e_n. But this time we continue by inserting e_{n+1}, changing the new representation state to $m.rep_{n+1}$ (bottom trajectory). Subsequent representation states may be different than for the first input sequence, so we mark them with double primes (") in place of single primes ('). But by the same arguments as above we conclude:

$$A(m.\boldsymbol{rep}_{0''}) = A(m.\boldsymbol{rep}_{n+1}) \tag{2a}$$

$$F_{n+1} = A(m.\boldsymbol{rep}_{0''}) \tag{2b}$$

By assumption, the `Insert` operation also can be verified, so it works correctly when we insert the edge e_{n+1} (the diagonal arrow in Fig. 4). From the postcondition of `Insert` with appropriate substitutions for that invocation, we therefore know:

$$IS\_MSF(A(m.\boldsymbol{rep}_{n+1}),\ A(m.\boldsymbol{rep}_n) \cup \{e_{n+1}\}) \tag{3}$$

3. We ignore the remainder of the correspondence, which trivally maps a flag indicating the machine's phase to the other component of the conceptual value, `m.insertion_phase`.

Now substituting in (3) from (1a), (1b) and (2a), (2b), we deduce:

$$IS\_MSF(F_{n+1}, F_n \cup \{e_{n+1}\}) \quad (4)$$

But if (4) holds then $I \in M$. We conclude, then, that every valid deterministic batch-style implementation of `Spanning_Forest_Machine` that can be proved using an abstraction function is monotonic. Yet we know there are valid and practical deterministic batch-style implementations that are nonmonotonic. A proof system that relies solely on abstraction functions, therefore, cannot be used to verify the correctness of any member of this entire class of correct and practical implementations of `Spanning_Forest_Machine_Template`.

3.4 Verification Using Abstraction Relations

The implementations discussed above have a natural and simple abstraction relation—the abstract value *m.edges* is any one of the MSFs of the graph stored in the internal representation *m.rep*. This relation, stated below formally using the predicate *IS_MSF*, is sufficient to prove the correctness of the implementations:

$$IS\_MSF(m.edges, S(m.rep))$$

where S is a function from the specific representation of machine m to the mathematical set of edges it contains. Details of the correctness proof are provided in Appendix B.

The MSF example shows that abstraction relations are essential for proving correctness of some implementations of nontrivial relational specifications. Such situations should be expected to arise in industrial-strength software systems. It is also possible that abstraction relations might be used—even though with effort they could be avoided in some cases—where they can simplify verification conditions and can be easier to understand than abstraction functions.

We note that a relational programming language semantics ultimately cannot be avoided if specifications are allowed to be relational—which they must be in order to permit specification of behavior such as that desired for `Spanning_Forest_Machine_Template` [7], [17]. Given that a relational semantics is essential, abstraction relations between concrete and abstract values do not increase verification complexity. For example, it is much easier to define the relation *IS_MSF* than the specific function computed by any given implementation, which depends on intricate algorithmic details involved in finding a particular MSF and which are inessential in the proof. Using an abstraction relation considerably simplifies the verification conditions for each of the `Spanning_Forest_Machine_Template` operations because the correspondence mapping is used separately in the verification of each operation [6], [7].

4 Discussion

The literature on verifying ADTs includes at least two conjectures that abstraction relations, even if technically required in some circumstances, are probably unnecessary in practice—at least where specifications are well-designed. When optimization problems are specified as procedures (for example, a simple operation for finding an MSF), this conjecture might be true. That situation demands relational specification of behavior and relational semantics, but not necessarily abstraction relations. The abstraction relation problem arises here because an optimization problem with possible ties has been captured not as a single procedure, but by recasting it as an ADT. In light of the advantages of the recasting approach [24], the abstraction relation issue assumes additional practical significance.

Formal systems that handle abstraction relations have been discussed (e.g., [9]) and some formal methods tool kits support them (e.g., Cogito [2]). But historically, abstraction functions have been so important and they are so entrenched that the generalization to abstraction relations tends to be resisted in some quarters. So we now examine (not necessarily published) attempts we have seen to avoid abstraction relations, and their ramifications.

The first approach is to prohibit the specification of relational behavior of operations. This would be undesirable when, as in this case, the natural intended behavior is inherently relational. Refusing to admit this possibility would leave a class of useful abstractions that could not be specified or that could not be easily reused in building other component implementations [7], [11], [13]. Moreover, specifying functional behavior for the MSF problem would rule out interesting classes of implementations and would make the specification much harder to understand—that specification would have to single out precisely which MSF must be produced, even in case of ties.

A second approach is to augment *m.rep* with an adjunct (auxiliary) variable, say *m.rep.abs*, which simply mirrors the abstract value. This would give a trivial abstraction function: $m = m.rep.abs$, and it would introduce a representation convention (invariant) relating *m.rep.abs* to the rest of *m.rep* (i.e., the original concrete representation). Notice that the availability of this approach does not refute our proof in Section 3 because the adjunct code required to update *m.rep.abs* would be nondeterministic, and any implementation written like this would not be in ***B***. Nonetheless it might be argued that any implementation in ***B*** *could* be transformed in this way in order to avoid abstraction relations.

Even if valid, this suggestion would have little practical import because following it would just move the expression of the required relation from the correspondence to the representation convention. Correctness proofs would not be simplified at all. But the bigger problem is that a batch implementation of `Spanning_Forest_Machine_Template` that employed this device would be incorrect. The adjunct code to update *m.rep.abs* in Insert would have to select a particular MSF prematurely (albeit nondeterministically), and subsequent `Extract` operations could not be proved to return precisely the edges of the selected MSF.

A third approach involves changing the specifications. This has been considered both in the ADT framework [11] and in related work [5] on concurrent processes involving I/O automata and sequences of actions. Lynch documents the practical utility of "multivalued possibilities mappings" (the I/O automata counterpart of abstraction relations) [16]. However, Abadi and Lam-

port show that specifications can be transformed to avoid multivalued mappings, under certain conditions; "refinement mappings" (the counterpart of abstraction functions) always exist [1]. Abadi and Lamport introduce techniques to avoid abstraction relations which, when adapted to the ADT framework, require changing the specifications of some of the components involved in the proof. Changes to these specifications are ruled out by the modularity requirements we place on ADT correctness proofs.

The issue of changing specifications of components does raise the question of whether the `Spanning_Forest_Machine_Template` specification could be designed differently to avoid the need for abstraction relations. For example, suppose that the specification is changed to be along the lines discussed in Section 3.4, i.e., conceptually all the edges are kept during the insertion phase, and an MSF is chosen only in `Change_To_Extraction_Phase`. Then for an implementation that mirrors this specification temporally, i.e., for a batch-style one that computes an MSF in `Change_To_Extraction_Phase`, an abstraction function is sufficient for a proof of correctness. However, the amortized-cost implementation in [24], and implementations that defer computation of an MSF to the `Extract` operation such as the one discussed in Section 2.2, still require an abstraction relation. Furthermore, if one must change specifications for the sake of avoiding abstraction relations in correctness proofs—without regard for the impact on understandability to potential component clients [20], [26]—then some of the most important practical benefits of formal specification for software engineering may be lost.

Even if a specification of `Spanning_Forest_Machine_Template` is devised that avoids the need for abstraction relations for that example [22], the completeness and naturalness needs raised in this paper remain. The practical requirement for abstraction relations to handle specifications that are not fully abstract will also continue to exist, because software developers are likely to continue to design and use such concepts. Fully embracing abstraction relations is therefore an essential practical step in broadening the applicability of formal methods beyond simplistic data abstractions.

Appendix A – RESOLVE Notation

The specifications in this paper are written in RESOLVE, a detailed description of which is available elsewhere [21]. Here we give a brief overview of RESOLVE notation that (along with a general understanding of issues in specification and implementation of abstract data types) should be sufficient to enable a reader to understand the examples in this paper.

A.1 Specification Notation

A RESOLVE **concept** specifies an "abstract template" (generic abstract module) by listing its **context**, which explicitly defines all coupling to the environment and makes all local declarations used in the rest of the specification; and its exported **interface**. The **global context** section identifies fixed coupling of this module to others in a shared component library. The **parametric context** section defines the ways in which a client can provide context, through parameters, when instantiating the generic module. The **local context** section typically introduces special-purpose mathematical notation used in the interface specification. For example, in Fig. 2, *IS_MSF* is a mathematical operation (function). Its **definition** should say formally that *IS_MSF*(*msf*, *g*) is true iff *msf* is a minimum spanning forest of *g*. We have elided this to focus on the more important features of the specification, but *IS_MSF* can be formally defined in a few lines.

The **interface** section explains the concept's exported types and operations. Each program type (family) is explained by referring to its mathematical model. For example, the type `Spanning_Forest_Machine` in Fig. 2 is modeled as an ordered pair consisting of a set of *EDGEs* and a boolean value. The **constraint** clause for a mathematical model (e.g., *EDGE* or the model for `Spanning_Forest_Machine`) says that, of all possible values of the underlying mathematical space, only those satisfying that clause are part of the model space.

Every program type has three implicit operations:

1) The **initialize** operation is invoked only at the beginning of the scope where its argument is declared. It gives the variable an initial value, which is specified in the **initialization ensures** clause of the type specification.
2) The **finalize** operation is invoked only at the end of the scope where its argument is declared, so usually there is no need to specify its abstract effect—because there is none. This operation is generally a hook for the type's implementer to release resources (e.g., memory) used by the representation.
3) The **swap** operation (invoked using the infix :=: operator) exchanges the values of its two arguments [8].

RESOLVE specifications never include preconditions like "*x* has been initialized" because client programs *always* initialize and finalize variables at the beginning and end of scope, respectively. In RESOLVE **initialize** and **finalize** work like C++ constructor and destructor operations, with appropriate calls generated by the compiler.

The effect of each operation is specified using a **requires** clause (precondition) and an **ensures** clause (postcondition). Each of these is an assertion about the values of the mathematical models of the operation's parameters. A missing clause means the assertion is the constant **true**. Mathematically, an operation defines a partial relation on the space of input and output values of the parameters: The **requires** clause tells where the relation is defined, and the **ensures** clause defines it there. Operationally, the contract between operation client and implementer is as follows: If a client calls an operation in a state in which the **requires** clause holds for the actual parameters, then the implementer guarantees that the operation will return in a state in which the **ensures** clause holds; but if the **requires** clause does not hold when the call occurs, then the implementer makes no guarantees whatsoever.

In a **requires** clause a variable stands for its value at the beginning of a call. In an **ensures** clause a variable stands for its value at the end of the call, while a variable with a prefixed # (pronounced "old") stands for the value of that variable at the beginning of the call. Other

mathematical notations depend on the mathematical theories involved in the explanation of behavior. The specification of `Spanning_Forest_Machine_Template` uses finite sets, tuples, integers, and booleans.

Operation specifications are considerably simplified by using abstract parameter modes **alters**, **produces**, **consumes**, and **preserves**. An **alters**-mode parameter potentially is changed by executing the operation; the **ensures** clause says how. A **produces**-mode parameter gets a new value that is defined by the **ensures** clause, which may *not* involve the parameter's old value because it is irrelevant to the operation's effect. A **consumes**-mode parameter gets a new value that is an initial value for its type, but its old value is relevant to the operation's effect. A **preserves**-mode parameter suffers no net change in value between the beginning of the operation and its return, although its value might be changed temporarily while the operation is executing.

A.2 Realization Notation

A RESOLVE **realization** describes a "concrete template" (generic implementation module). A **facility** is an instance of a concept with an associated instance of a realization which implements it, so its declaration involves choosing and fixing the parameters of both a concept and one of that concept's realizations. In operation bodies, the representation of a variable (e.g., *m*) of an exported type is designated as *m.rep* so it is clear that this is the representation model's value and not the conceptual model's value.

RESOLVE realization code contains three kinds of assertions. For every loop there is a loop invariant or loop specification; and for every type there is a **convention** assertion that characterizes the subspace of representation configurations that might arise (the representation invariant), and a **correspondence** assertion that explains how to associate such representation configurations with conceptual model values (the abstraction relation).

The **convention** clause of Fig. 5 uses the built-in RESOLVE mathematical function **elements**, which denotes the set of entries in the string of items which is its argument. So, if *str* = <a, b, c, b>, then **elements** (*str*) = {a, b, c}.

Appendix B – Verification of a Batch-Style Implementation

In this appendix we prove the correctness of the batch-style implementation of `Spanning_Forest_Machine_Template` shown in Fig. 5. Its global context section refers to `Queue_Template`, which is shown in Fig. 6 for completeness.

A proof of correctness [7] of the realization of Fig. 5 starts by factoring out two lemmas that arise during the verification of each individual operation:

C1. For every representation state for which the **convention** clause holds, there is a conceptual state to which the **correspondence** clause relates it.

C2. For every representation state for which the **convention** clause holds, and for every conceptual state related to it by the **correspondence** clause, the **constraint** clause (see Fig. 2) holds for the conceptual state.

In this case, to prove C1 we must prove that there is at least one conceptual `Spanning_Forest_Machine` value for every Machine_Rep value that can arise. This follows from the definition of MSF in graph theory (which we assume is encoded formally in the predicate *IS_MSF*); i.e., every graph has an MSF. To prove C2 we must prove that any MSF of any graph is its own MSF; and again this is a simple lemma in graph theory.

The verification is completed by showing that for each operation body, the code implements the associated abstract operation specification. For each operation and for each fixed assignment of values to all the other arguments, we consider four sets of values for each `Spanning_Forest_Machine` argument: initial and final conceptual states, C_i and C_f, respectively; and initial and final representation states, R_i and R_f, respectively. R_i contains those representation states for which: 1) the **convention** clause holds, 2) there exists a conceptual state satisfying the **correspondence** clause and this particular operation's abstract precondition, and 3) every such corresponding conceptual state satisfies this operation's precondition. R_f contains the representation states that can be reached from some representation state in R_i by correct execution of the operation's body. C_i and C_f contain the conceptual states for which the **correspondence** clause holds for some representation state in R_i and R_f, respectively.

Informally stated, we have three kinds of proof obligations; i.e., the verification conditions are of these three forms:

V1. For every $r \in R_i$ and for every trajectory leading from r through the operation body, all internal assertions (e.g., loop invariants) hold at the appropriate times, and the preconditions of all called operations hold at the points they are called. (This obligation arises from the need to define R_f, since only if a called operation's precondition holds may we assume that its effect is what we expect from its specified postcondition.)

V2. For every $r \in R_f$ the **convention** clause holds. (This obligation arises from the need to define C_f, since only in this case is it certain that there is some conceptual state to which the **correspondence** clause relates every $r \in R_f$.)

V3. For every $\#r \in R_i$, $r \in R_f$, and $c \in C_f$ for which r is reachable from $\#r$ by some correct execution of the operation body and where the **correspondence** clause relates c and r, there exists some $\#c \in C_i$ for which the **correspondence** clause relates $\#c$ and $\#r$ and where the operation's abstract postcondition holds for $\#c$ and c. (This obligation arises from the need to complete the "commutativity diagram" [7, pp. 303-305].)

There also is a specialized version of such a proof for the **initialize** operation, where we may neither assume that the **convention** clause holds for the initial representation state nor, consequently, that there is any initial conceptual state corresponding to it.

In this case, it is easy to discharge the obligations of the forms V1 and V2 for each operation. There is only

```
realization body Batch for Spanning_Forest_Machine_Template

   context

      global context
         concept Record3_Template
         concept Queue_Template
         concept Record2_Template
         facility Standard_Boolean_Facility

      local context

         type Edge is record
            v1: Integer
            v2: Integer
            e: Integer
         end record

         facility Edge_Queue_Facility is Queue_Template(Edge)
                     realized by Queue_Realization_1

         operation Produce_MSF (
               alters q: Edge_Queue_Facility.Queue
            )
            ensures
              IS_MSF (elements (q), elements (#q)) and
              |q| = |elements (q)|
         begin
            -- code that batch computes an arbitrary MSF of q
         end Produce_MSF

   interface
      type Spanning_Forest_Machine is represented by record
         graph_edges: Edge_Queue_Facility.Queue
         insertion_flag: Boolean
      end record
         convention
            if not m.rep.insertion_flag
            then IS_MSF (elements (m.rep.graph_edges),
                         elements (m.rep.graph_edges))
         correspondence
            IS_MSF (m.edges, elements (m.rep.graph_edges)) and
            m.insertion_phase = m.rep.insertion_flag

      operation initialize
         begin
            m.rep.insertion_flag := true
         end initialize

      operation Change_To_Insertion_Phase (
              alters      m: Spanning_Forest_Machine
            )
         local context
            variables
              new_rep: Spanning_Forest_Machine
         begin
           m.rep :=: new_rep
           m.rep.insertion_flag := true
         end Change_To_Insertion_Phase

      operation Insert (
              alters      m: Spanning_Forest_Machine
              consumes    v1: Integer
              consumes    v2: Integer
              consumes    w: Integer
            )
         begin
           Enqueue (m.rep.graph_edges, (v1, v2, w))
         end Insert
```

```
      operation Change_To_Extraction_Phase (
              alters          m: Spanning_Forest_Machine
          )
        begin
          Produce_MSF (m.rep.graph_edges)
          m.rep.insertion_flag := false
        end Change_To_Extraction_Phase

      operation Extract (
              alters          m: Spanning_Forest_Machine
              produces        v1: Integer
              produces        v2: Integer
              produces        w: Integer
          )
        begin
          Dequeue (m.rep.graph_edges, (v1, v2, w))
        end Extract

      operation Size (
              preserves       m: Spanning_Forest_Machine
          ): Integer
        begin
           return Length (m.rep.graph_edges)
        end Size

      operation Is_In_Insertion_Phase (
              preserves m: Spanning_Forest_Machine
          ): Boolean
        begin
          return m.rep.insertion_flag
        end Is_In_Insertion_Phase

end Batch
```

Fig. 5. Realization body for a batch-style implementation.

```
concept Queue_Template

   context

      global context
         facility Standard_Integer_Facility

      parametric context
         type Item

   interface

      type Queue is modeled by string of math[Item]
         exemplar q
         initialization ensures
            q = empty_string

      operation Enqueue (
              alters          q: Queue
              consumes        x: Item
          )
         ensures
            q = #q * <#x>

      operation Dequeue (
              alters          q: Queue
              produces        x: Item
          )
         ensures
           #q = <x> * q

      operation Length (
              preserves       q: Queue
```

```
    ): Integer
  ensures
    Length = |q|

end Queue_Template
```

Fig. 6. Specification of Queue_Template.

one called operation (Dequeue in the body of Extract) that has a nontrivial precondition, and in this case the precondition of Extract implies that R_i contains only representation states where *m.rep.graph_edges* is not empty. Showing that the **convention** clause holds at the end of each operation body is more tedious because it involves *m.rep.insertion_flag* as well as *m.rep.graph_edges*, but the details are straightforward.

All the proof obligations of the form V3 are similarly trivial, except for the Insert operation. Here, the proof of the only troublesome part of the applicable verification condition follows directly from the graph theory lemma stated in Section 3.

We observe that the verification of this batch implementation answers the question posed in Section 3.1.1: Can the edges obtained from a series of Extract operations be *any* MSF of the edges that were inserted into a Spanning_Forest_Machine? The body of procedure Produce_MSF in Fig. 5 may produce *any* MSF of the edges it is given. The realization in Fig. 5 is correct with no further assumptions about which MSF that must be. So the specification of Spanning_Forest_Machine_Template truly places no restriction on which MSF of the inserted edges might be produced.

Acknowledgments

We thank Anish Arora, Steve Edwards, David Fleming, Wayne Heym, Joe Hollingsworth, Chip Klostermeyer, Tim Long, Sethu Sreerama, and Stu Zweben for their insightful comments on drafts of this paper; and Doug Kerr, Nathan Loofbourrow, Spiro Michaylov, and Tom Page for helpful discussions on some key technical points. Comments from the anonymous referees have helped clarify several issues.

Murali Sitaraman was sponsored by the National Science Foundation (NSF) under grant CCR-9204461; Advanced Research Projects Agency (ARPA) under contract numbers DAAH04-96-1-0419 and DAAH04-94-G-0002, both monitored by the U.S. Army Research Office; and the National Aeronautics and Space Administration (NASA) under grant 7629/229/0824.

Bruce W. Weide and William F. Ogden were sponsored by NSF under grant number CCR-9311702 and by ARPA under contract number F30602-93-C-0243, monitored by the USAF Materiel Command, Rome Laboratories, ARPA order number A714.

References

[1] M. Abadi and L. Lamport, "The Existence of Refinement Mappings," *Theoretical Computing Science*, vol. 82, no. 2, pp. 253–284, May 1991.

[2] A. Bloesch, E. Kazmierczak, P. Kearney, and O. Traynor, "Cogito: A Methodology and System for Formal Software Development," *Intl. J. Software Eng. and Knowledge Eng.*, vol. 5, no. 4 pp. 599–617, Dec. 1995.

[3] S.A. Cook, "Soundness and Completeness of an Axiom System for Program Verification," *SIAM J. Computing*, vol. 7, no. 1, pp. 70–90, Feb. 1978.

[4] T.H. Cormen, C.E. Leiserson, and R.L. Rivest, *Introduction to Algorithms*. Cambridge, Mass.: MIT Press, 1990.

[5] H.-D. Ehrich and A. Sernadas, "Algebraic Implementation of Objects over Objects," *Stepwise Refinement of Distributed Systems—Lecture Notes in Computer Science 430*. New York: Springer-Verlag, pp. 239–266, 1990.

[6] G.W. Ernst, R.J. Hookway, J.A. Menegay, and W.F. Ogden, "Modular Verification of Ada Generics," *Computer Language*, vol. 16, nos. 3/4, pp. 259–280, 1991.

[7] G.W. Ernst, R.J. Hookway, and W.F. Ogden, "Modular Verification of Data Abstractions with Shared Realizations," *IEEE Trans. Software Eng.* vol. 20, no. 4, pp. 288–307, Apr. 1994.

[8] D.E. Harms and B.W. Weide, "Copying and Swapping: Influences on the Design of Reusable Software Components," *IEEE Trans. Software Eng.*, vol. 17, no. 5, pp. 424–435, May 1991.

[9] J. He, C.A.R. Hoare, and J.W. Sanders, "Data Refinement Refined," *Lecture Notes in Computer Science 213*, B. Robinet and R. Wilhelm, eds., pp. 187–196, Springer-Verlag.

[10] C.A.R. Hoare, "Proof of Correctness of Data Representations," *Acta Informatica*, vol. 1, no. 1, pp. 271–281, 1972; also in D. Gries, ed., *Programming Methodology: A Collection of Articles by Members of IFIP WG 2.3*, pp. 269–281, New York: Springer-Verlag, 1978.

[11] C.B. Jones, *Systematic Software Development Using VDM*, Hertfordshire, U.K.: Prentice Hall Int'l, 1990.

[12] D. Kapur and S. Mandayam, "Expressiveness of the Operation Set of a Data Abstraction," *Conf. Record Seventh Ann. ACM Symp. Principles of Programming Languages*, pp. 139–153, ACM, 1980.

[13] G.T. Leavens, "Modular Specification and Verification of Object-Oriented Programs," *IEEE Software*, vol. 8, no. 4, pp. 72–80, July 1991.

[14] B. Liskov and J.M. Wing, "A New Definition of the Subtype Relation," *ECOOP 1993—Lecture Notes in Computer Science 707*, pp. 118–141. New York: Springer-Verlag, 1993.

[15] B. Liskov and J.M. Wing, "Corrigenda to ECOOP '93 Paper," *ACM SIGPLAN Notices*, vol. 29, no. 4, p. 4, Apr. 1994.

[16] N. Lynch, "Multivalued Possibilities Mappings," *Stepwise Refinement of Distributed Systems—Lecture Notes in Computer Science 430*, pp. 519–543, New York: Springer-Verlag, 1990.

[17] G. Nelson, "A Generalization of Dijkstra's Calculus," *ACM Trans. on Program Languages and Systems*, vol. 11, no. 4, pp. 517–561, Oct. 1989.

[18] T. Nipkow, "Are Homomorphisms Sufficient for Behavioral Implementations of Deterministic and Nondeterministic Data Types?" *Lecture Notes in Computer Science 247*, F.J. Brandenburg, G. Vidal-Naquet, and M. Wirsing, eds., pp. 260–271, New York: Springer-Verlag, 1987.

[19] O. Schoett, "Behavioral Correctness of Data Representations," *Science of Computer Programming*, vol. 14, pp. 43–57, 1990.

[20] M. Sitaraman, L.R. Welch, and D.E. Harms, "On Specification of Reusable Software Components," *Intl. J. Software Eng. and Knowledge Eng.*, vol. 3, no. 2 , pp. 207–219, June 1993.

[21] "Special Feature: Component-Based Software Using RESOLVE," M. Sitaraman and B.W. Weide, eds., *ACM SIGSOFT Software Eng. Notes*, vol. 19, no. 4, pp. 21–67, Oct. 1994.

[22] M. Sitaraman, "Impact of Performance Considerations on Formal Specification Design," *Formal Aspects of Computing*, vol. 8, no. 6, pp. 716–736, 1997.

[23] B.W. Weide and J.E. Hollingsworth, "Scalability of Reuse Technology to Large Systems Requires Local Certifiability," *Proc. Fifth Ann. Workshop Software Reuse*, Palo Alto, Calif., Oct. 1992.

[24] B.W. Weide, W.F. Ogden, and M. Sitaraman, "Recasting Algorithms to Encourage Reuse," *IEEE Software*, vol. 11, no. 5 pp. 80–88, Sept. 1994.

[25] B.W. Weide, S.H. Edwards, W.D. Heym, T.J. Long, and W.F. Ogden, "Characterizing Observability and Controllability of Software Components," *Proc. Fourth Int'l Conf. Software Reuse*, M. Sitaraman, ed., pp. 62–71, IEEE, Apr. 1996.
[26] J.M. Wing, "A Specifier's Introduction to Formal Methods," *Computer*, vol. 23, no. 9, pp. 8–24, Sept. 1990.

Murali Sitaraman received his ME in computer science from the Indian Institute of Science, Bangalore, and his PhD in computer and information science from The Ohio State University. He is presently an associate professor of computer science at West Virginia University. Sitaraman is a principal investigator of the RESOLVE research effort and directs the Reusable Software Research Group at WVU (http://www.cs.wvu.edu/~resolve). His interests span theoretical, practical, and educational aspects of software engineering. Sitaraman served as program chair of the *Fourth International Conference on Software Reuse* (Orlando, Florida, April 1996). He is a member of the IEEE Computer Society and ACM.

Bruce W. Weide holds a PhD in computer science from Carnegie Mellon University and a BSEE from the University of Toledo. He is professor of computer and information science at The Ohio State University, where he co-directs the Reusable Software Research Group with Bill Ogden and Stu Zweben. His research interests include all aspects of software component engineering, especially in applying RSRG work to practice and in teaching its principles to beginning CS students. He is a member of the IEEE, ACM, and CPSR.

William F. Ogden received his MS and PhD degrees from Stanford University, and his BS degree from the University of Arkansas. He is an associate professor of computer and information science at The Ohio State University and co-director of the Reusable Software Research Group with Bruce Weide and Stu Zweben. Previously, he served on the faculties at Case Western Reserve University and the University of Michigan. His main research interests are in software reuse, software specification, and program verification. He is a member of the IEEE Computer Society and ACM.

Eighth Annual Workshop on Software Reuse

March 23–26, 1997
Columbus, Ohio

INDIANA UNIVERSITY SOUTHEAST

UNIVERSITY OF MAINE

Virginia Tech

Software Component Relationships

Stephen H. Edwards

Department of Computer Science
Virginia Polytechnic Institute and State University
660 McBryde Hall
Blacksburg, VA 24061-0106
Tel: (540)-231-7537
Email: edwards@cs.vt.edu

David S. Gibson, Bruce W. Weide, and Sergey Zhupanov

Department of Computer and Information Science
The Ohio State University
2015 Neil Avenue
Columbus, OH 43210
Tel: (614) 292-5813
Email: {dgibson|weide|sergey}@cis.ohio-state.edu

Abstract

Large complex software systems are composed of many software components. Construction and maintenance of component-based systems require a clear understanding of the dependencies between these components. To support reuse, components should be designed to minimize such dependencies. When component coupling is necessary, however, dependencies need to be expressed clearly and precisely. Most software analysis and design methodologies rely on relationships such as *passes-data-to*, *calls*, *is-a-part-of*, and *inherits-from* for this purpose. Our position is that component relationships such as these are not an effective way to convey important dependency information to implementors and maintainers working with reusable software components. Precisely-defined conceptual relationships are better suited to this task.

Keywords: Software components, component relationships, software engineering, software reuse, behavioral substitutability

Workshop Goals: Learning, feedback, networking

Working Groups: (1) Rigorous Behavioral Specification as an Aid to Reuse, (2) Component Certification Tools, Frameworks and Processes, and (3) Object Technology, Architectures, and Domain Analysis: What's the Connection? Is There a Connection?

1 Background

As members of the Reusable Software Research Group (RSRG) at The Ohio State University, we have been exploring various aspects of software component engineering for over ten years. One recent focus of our research has been on the identification, formalization, and expression of dependency relationships between software components. Many of our ideas on software component relationships are based on the results of RSRG research which has been incorporated into the RESOLVE framework, language, and discipline for software component engineering [1]. Newer ideas and terminology for expressing these relationships are based on a formal model of software subsystems called ACTI, for "Abstract and Concrete Templates and Instances[2].

To demonstrate the application of our ideas on software component relationships, we have developed the RESOLVE/C++ and RESOLVE/Ada95 disciplines for software component engineering. Steve Edwards, Bruce Weide, and Sergey Zhupanov developed the RESOLVE/C++ discipline. David Gibson developed the RESOLVE/Ada95 discipline. These disciplines use the language mechanisms of C++ and Ada (as revised in 1995) to encode language-independent software component relationships. Bruce Weide is currently using the RESOLVE/C++ discipline in the introductory Computer Science course sequence at Ohio State.

2 Position

Building software systems from reusable software components has long been a goal of software engineers. While other engineering disciplines successfully apply the reusable component approach to build physical systems, it has proven more difficult to apply in software engineering. A primary reason for this difficulty is that distinct software components tend to be more tightly coupled with each other than most physical components. Furthermore, components are often designed with extremely subtle dependencies on other components are which are not explicitly described. These dependencies may significantly complicate reasoning about program behavior[3].

Clearly some dependencies between software components are necessary and desirable. These dependencies need to be clearly expressed by component designers and well-understood by implementors and maintainers. The role of software component relationships is to express dependencies between components and, in doing so, to provide information about how components may and should be used in conjunction with other components. *Our position is that the software component relationships used by most analysis and design methodologies are not well-suited for building and maintaining large complex systems and that there are more suitable alternatives.*

Commonly used component relationships suffer from one or more of the following problems:

- they only describe *particular* component compositions, not what compositions are *possible*,
- they express *vague* meanings, which are of limited use, and
- they reflect *language-specific* views of component composition rather than a *conceptual* view.

Traditional techniques based on functional decomposition of systems have software "part" relationships depicted in notations such as data flow diagrams and structure charts. Relationships such as *passes-data-to*, *calls*, and *is-a-part-of* are common to these notations. These relationships may

be useful for understanding how components of a particular system are related. However, they do not directly address how one might reuse individual components to build a new system or modify an existing system. That is, these notations do not describe the dependencies of a component independent of a particular use of that component.

Object-oriented analysis, design, and programming methodologies usually rely heavily on the *inherits-from* component (class) relationship. Unlike those described above, this relationship can describe component dependencies independent of a specific component (class instance) usage. However, *inherits-from* typically fails to convey precise useful information about component dependencies. Even if the vague and varied meanings of *inherits-from* play a useful role during analysis and design, this relationship is much less useful when working with specific components during implementation and maintenance. Inheritance is a programming language mechanism that can be used to encode conceptual relationships between software components. Inheritance is not itself a conceptual relationship.

Some people in the object-oriented community argue that inheritance should only be used to express the *is-a* relationship between two components. Unlike *inherits-from*, *is-a* is a conceptual relationship between components. However, the *is-a* relationship does not have a single precisely-defined meaning. Furthermore, even when formally defined in terms of *behavioral substitutability*, the *is-a* relationship generally does not convey specific design intent. For example, stating that component X *is-a* Y does not suggest any information about why X was designed to be substitutable for Y and thus how X might be used.

To address the problems described above, component relationships should:

- concisely express dependencies describing *possible* component compositions,
- have *precise* meanings useful to clients, implementors, and maintainers, and
- reflect a clear *conceptual* view of component-based software engineering.

We have defined a small set of language-independent component relationships which satisfy these three criteria. Our relationships describe the dependencies between components at a conceptual level. The relationships provide implementers and maintainers precise information about how components may and should be used. Furthermore, they may be used to express specific design intent. In the remainder of this section, we briefly introduce the component relationships: *implements*, *uses*, and *extends*. While we have defined several other useful component relationships, these are the most general and easiest to understand.

The software component relationships we have defined relate components which may either be abstract (specifications) or concrete (implementations)[1]. An *abstract component* describes functional behavior—*what* services a subsystem provides. A *concrete component* describes an implementation—*how* a subsystem's services are provided. Having separate abstract and concrete components supports data abstraction, information hiding, multiple implementations, and self-contained descriptions of component behavior.

The most fundamental component relationship upon which all others rely for meaning and utility is *implements*. *Implements* describes the key relationship between an implementation, a concrete

[1] In addition to the abstract versus concrete dimension, components are either templates or instances. These two orthogonal dimensions give rise to four kinds of components: abstract templates, abstract instances, concrete templates, and concrete instances. While the relationships introduced in this paper only deal with the abstract versus concrete dimension, we have identified template-specific relationships.

component, and a specification, an abstract component. The *implements* relationship may be defined informally as follows:

> Concrete component X *implements* abstract component Y if and only if X exhibits the behavior specified by Y.

This is a fairly intuitive relationship between a specification and an implementation. However, a fully formal and general definition of the *implements* relationship is very intricate and has been the subject of much research. *Implements* expresses a dependency between two components in the following sense. If component X *implements* component Y, then X depends on Y to provide an abstract, client-level description of its behavior – a "cover story" hiding *all* implementation details.

While justifying a claim that X *implements* Y may require significant effort, especially if done formally, considerable leverage is gained from doing so. If two different concrete components both implement the same abstract component, then either of them may be used in a context requiring the functional behavior described by the abstract component. In this case, the two implementations are *behaviorally substitutable* with respect to the specification they both implement. The two implementations may differ in non-functional characteristics such as execution time, space requirements, cost, warranty, legal use restrictions, level of trust in correctness, and so forth.

The *implements* relationship describes a dependency between an implementation and a specification. The *uses* relationship may describe a dependency that exists between two different abstractions. The relation name *uses* actually applies to three different yet closely related component relationships. *Uses* may describe a dependency between two implementations, between two specifications, or between an implementation and an specification. The last of these three relationships is defined as follows:

> Concrete component X *uses* abstract component Y if and only if X depends on the behavior specified by Y.

This form of the *uses* relationship expresses a *polymorphic* relationship between implementations. Any component that *implements* abstract component Y may serve as the actual concrete component used by instances of component X. Thus, this form of *uses* reduces unnecessary dependencies between components. Note that none of the three *uses* relationships is equivalent to the *is-a-part-of* relationship. If implementation X *uses* implementation Y, Y may or may not be a part of the data representation of X. The client wishing to use component X does not need to know Y's specific role in X, just that component Y is required in order to use component X.

A third component relationship is *extends*. The name *extends* applies to two different, yet closely related, component relationships. One *extends* expresses a dependency between two abstract components. The other expresses a dependency between two concrete components. Both *extends* relationships may be defined informally as follows:

> Component X *extends* component Y if and only if all of the interface and behavior described by Y is included in the interface and behavior described by X.

This definition conveys the intuitive meaning of *extends*, that is, component X extends the interface and behavior of component Y. It implies the essential property of behavioral substitutability. If

abstract component X *extends* abstract component Y, concrete component $X1$ *implements* X, and concrete component $Y1$ *implements* Y, then $X1$ is behaviorally substitutable for $Y1$ with respect to Y. Note that $Y1$ is *not* behaviorally substitutable for $X1$ with respect to X in this case. Thus behavioral substitutability is a ternary relationship, not a binary equivalence relation.

To some readers, the *extends* relationship may sound very much like an inheritance relationship. It is important to understand that *extends* is *not* an inheritance relationship. *Extends* describes a *behavioral* relationship between two components. *Inherits-from* does not. Furthermore, while inheritance may be a convenient programming language mechanism for expressing structural aspects of the *extends* relationship, *extends* may be encoded in programming languages without any use of inheritance.

3 Comparison

Perhaps the most widely known work which includes definitions of software component relationships is that by Grady Booch, Ivar Jacobson, and James Rumbaugh on the Unified Modeling Language (UML). The UML includes software component relationships in the form of class relationships defined in Booch's method for object-oriented analysis and design [4]. Booch identifies three basic kinds of class relationships: those expressing *is-a* relationships, those expressing *is-a-part-of* relationships, and *association* relationships which denote some semantic dependency between otherwise unrelated classes. The specific class relationships used by Booch include *association*, *inheritance*, *aggregation*, and *using*. The meanings of these relationships are largely influenced by available programming language mechanisms (in particular, those of C++).

Booch's *association* relationship is primarily useful for design and analysis of a *particular* application or composition of components. It does not convey information about what compositions are possible for a reusable component. Booch's use of the *inheritance* relationship is limited to expressing *is-a* relationships. However, his definition of *is-a* is not strict enough to imply behavioral substitutability with respect to some specification (as does *implements*). Thus the component relationships expressed as class relationships in the UML appear to suffer from all of the problems described in the last section.

Some object-oriented programming advocates such as Bertrand Meyer do not insist that inheritance only be used to express the conceptual *is-a* relationship. Meyer has described twelve different component relationships that may be expressed using inheritance, only one of which expresses the *is-a* relationship [5]. As with the UML's use of inheritance, Meyer's *is-a* use of inheritance is not defined precisely enough to imply behavioral substitutability.

Some researchers have studied precisely defined class relationships which do imply behavioral substitutability of components [6, 7]. This research largely focuses on how the inheritance language mechanism can and should be used in a manner that supports reasoning about program behavior. Unlike our research, this work does not address conceptual component relationships from a language-independent perspective.

References

[1] M. Sitaraman and B. W. Weide, editors, "Special feature: Component-based software using RESOLVE," *ACM SIGSOFT Software Engineering Notes*, vol. 19, no. 4, pp. 21–67, 1994.

[2] S. H. Edwards, *A Formal Model of Software Subsystems.* PhD thesis, The Department of Computer and Information Science, The Ohio State University, Columbus, Ohio, 1995.

[3] B. W. Weide and J. E. Hollingsworth, "Scalability of reuse technology to larege systems requires local certifiability," in *Proceedings of the Fifth Annual Workshop on Software Reuse* (L. Latour, ed.), Oct. 1992.

[4] G. Booch, *Object-Oriented Analysis and Design With Applications.* Menlo Park, CA: Benjamin/Cummings, 2nd ed., 1994.

[5] B. Meyer, "The many faces of inheritance: A taxonomy of taxonomy," *IEEE Computer*, vol. 29, pp. 105–108, May 1996.

[6] B. H. Liskov and J. M. Wing, "A behavioral notion of subtyping," *ACM Transactions on Programming Languages and Systems*, vol. 16, pp. 1811–1841, Nov. 1994.

[7] K. K. Dhara and G. T. Leavens, "Forcing behavioral subtyping through specification inheritance," in *Proceedings of the 18th International Conference on Software Engineering*, pp. 258–267, IEEE Computer Society Press, Mar. 1996.

Biography

Stephen Edwards is a Visiting Assistant Professor in the Department of Computer Science at the Virginia Polytechnic Institute and State University. He received a B.S. in Electrical Engineering at the California Institute of Technology in 1988 and his M.S. and Ph.D in Computer and Information Science at The Ohio State University in 1992 and 1995, respectively.

David Gibson is a Major in the United States Air Force and a Ph.D. candidate in the Department of Computer and Information Science at The Ohio State University. He received a B.S. in Physics and Computer Science from Duke University in 1983 and his M.S. in Computer and Information Sciences from Trinity University in San Antonio in 1986.

Bruce Weide is a Professor of Computer and Information Science at The Ohio State University. He received a B.S. in Electrical Engineering from the University of Toledo in 1974 and a Ph.D. in Computer Science from Carnegie-Mellon University in 1978. He is a co-founder of the Reusable Software Research Group at The Ohio State University.

Sergey Zhupanov is a Research Associate in the Department of Computer and Information Science at The Ohio State University. He received his B.S. and M.S in Computer and Information Science at The Ohio State University in 1994 and 1996, respectively.

We gratefully acknowledge financial support from the National Science Foundation (grant number CCR-9311702, DUE-9555062, and CDA-9634425), the Advanced Research Projects Agency (contract number F30602-93-C-0243, monitored by the USAF Materiel Command, Rome Laboratories, ARPA order number A714), and the Fund for the Improvement of Post-Secondary Education under project number P116B60717. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the National Science Foundation, the Defense Advanced Research Projects Agency, or the U.S. Department of Education.

Reverse Engineering of Legacy Code Exposed[1]

Bruce W. Weide and Wayne D. Heym
Department of Computer and Information Science
The Ohio State University, Columbus, OH 43210
{weide,heym}@cis.ohio-state.edu

Joseph E. Hollingsworth
Department of Computer Science
Indiana University Southeast, New Albany, IN 47150
jholly@ius.indiana.edu

Abstract — Reverse engineering of large legacy software systems generally cannot meet its objectives because it cannot be cost-effective. There are two main reasons for this. First, it is very costly to "understand" legacy code sufficiently well to permit changes to be made safely, because reverse engineering of legacy code is intractable in the usual computational complexity sense. Second, *even if* legacy code could be cost-effectively reverse engineered, the ultimate objective — re-engineering code to create a system that will not need to be reverse engineered again in the future — is presently unattainable. Not just crusty old systems, but even ones engineered today, from scratch, cannot escape the clutches of intractability until software engineers learn to design systems that support modular reasoning about their behavior. We hope these observations serve as a wake-up call to those who dream of developing high-quality software systems by transforming them from defective raw materials.

1. Introduction

Most large software systems, even if apparently well-engineered on a component-by-component basis, have proved to be incoherent as a whole due to unanticipated long-range "weird interactions" among supposedly independent parts. The best anecdotal evidence for this conclusion comes from reported experience dealing with *legacy* code, i.e., programs[2] in which too much has been invested just to throw away but which have proved to be obscure, mysterious, and brittle in the face of maintenance.

What should we do when we require a new system whose behavior is intended to be similar to that of an old system we already have? One option is to build the new one from scratch, relying perhaps on experience obtained through design or use of the old one, but not relying substantially on the old code. Another option is to try to understand the old code well enough to keep much of it, modifying it to meet the new needs. The latter approach — *re-engineering* — necessarily involves *reverse engineering*:

> Reverse engineering encompasses a wide array of tasks related to understanding and modifying software systems. Central to these tasks is identifying the components of an existing software system and the relationships among them. Also central is creating high-level descriptions of various aspects of existing systems. [15, p. 23]

We consider reverse engineering in its role as an integral part of the re-engineering approach to new system development. The objective of reverse engineering is not (just) to create documents that chronicle a path from the original requirements to the present legacy system as, say, a substitute for the documentation that probably was not created while that journey was in progress. The goal is to achieve a sufficient understanding of the whats, hows, and whys of the legacy system as a whole that its code can be re-engineered to meet new requirements on behavior, performance, structure, system dependencies, etc.

1.1. Reverse Engineering of Legacy Code is Intractable

There seems to be general agreement that, in practice, reverse engineering of legacy code is at least quite laborious [14]. Even if many aspects of large systems are easy to understand, inevitably there is important behavior whose explanation is latent in the code yet which resolutely resists discovery. The basic reason is that software engineers seek modularity — and they generally achieve it well enough create a very compact representation of system behavior in the source code, but not well enough to support modular reasoning about that behavior. In Sections 3-4 we summarize how this implies that reverse engineering of legacy code is intractable in the usual computational complexity sense [19]. This fundamental conclusion and the supporting argument follow up on a suggestion by Hopkins and Sitaraman [9] that the effort required to reverse engineer a system is related to the effort required to formally verify its functional correctness. In fact, if we argued that program verification of legacy code is intractable, there probably would be little debate (at least with those from whom the current position is likely to draw fire). Yet these are technically equivalent.

---

1 This position paper is adapted from [0,19].

2 We do not consider legacy systems that consist primarily of data (e.g., databases).

ICSE '95, Seattle, Washington USA

1.2. *Forward Engineering is Not a Solved Problem*

Of course intractable does not mean impossible. One even hears occasional stories about "successful" reverse and re-engineering projects [1]. These should be taken with a grain of salt, if only because the real problem — successful re-engineering — cannot be known to have been solved for years, after there is a long history of maintenance tasks to sort through or the system has to be re-engineered again. Even then, without a controlled study, there is no way to know that building from scratch would not have been more cost-effective. And even if the reverse engineering battle can be won once, the re-engineering war ultimately will be lost without subsequent use of forward engineering techniques that effectively prevent software "rot".

Unfortunately, almost without exception software engineers do not know how to design and build truly modular systems when starting from scratch, let alone when starting from legacy code [17, 18]. Except for egregiously poor design practices, they cannot distinguish fair-to-good software designs from excellent ones. The reason is that beyond "structured programming" aphorisms there are almost no accepted community standards for what software systems should be like at the detail level. By the intractability argument, some key quality criteria would seem to be understandability in general, and susceptibility to modular reasoning about behavior in particular. Yet this degree of modularity is almost universally *not* achieved by designs in computer science textbooks, technical papers, and commercial software.

2. Observations and Implications

Reverse engineering of legacy code has proved to be such a difficult practical problem — experience which lends credence to the thesis that it is intractable — that serious attention ought to be devoted to the subject. This is particularly true because the alternative is also costly. But we need to have realistic expectations about the ultimate role of reverse engineering in a comprehensive vision of software engineering. We are disturbed with the emphasis on building tools to solve problems whose inherent complexity suggests that those tools cannot be expected to scale up to realistically large systems. And we are frankly alarmed by the following sentiments, which seem typical among reverse engineering advocates:

> ... while many of us may dream that the central business of software engineering is creating clearly understood new systems, the central business is really upgrading poorly understood old systems. [15, p. 23]
>
> [The problem of having to deal with] legacy software is basically the result of management inaction rather than technology deficiency... [1, p. 23]

Quite the contrary! Most software hasn't been written yet, and widely taught and practiced "modern" approaches to the nuts-and-bolts of software engineering still do *not* lead to well-designed modular systems. So, if we as a community act as though we believe that "the central business [of software engineering] is really upgrading poorly understood old systems," then we will squander a fortune yet continue to face the Sisyphean task of upgrading poorly understood old systems into slightly less poorly understood new systems. We might have spent our efforts developing and exploring truly productive techniques for forward engineering of well-understood modular systems — and this progress would help even those who insist that re-engineering is ultimately where the action will be.

We mentioned above that reverse engineering is as hard as program verification, and this leads to a common misunderstanding about the claim of intractability. General program verification is in principle unsolvable, because the verification conditions generated from code and specifications might include arbitrary mathematical assertions. The practical consequences of this observation, however, are minimal. It can be used to show that *there exist* esoteric systems for which program verification (hence reverse engineering) is *impossible*; but it does not mean that program verification/reverse engineering cannot succeed on code that arises in practical situations. Our claim is about such practical situations. Specifically, *for all* large legacy systems, program verification/reverse engineering is *prohibitively expensive*; not impossible in principle but manifestly not cost-effective — and this bears directly on the business decision regarding whether to re-engineer or to build anew. The obvious rejoinder to this claim from reverse engineering advocates is, "People do reverse engineering all the time; how can it be prohibitively expensive?" We address this question in Section 3.1.

The news is not all bad; we offer some assistance to reverse engineering advocates. Specifically, by identifying threats to modular reasoning from common design and coding practices as a key technical factor that thwarts cost-effective reverse engineering, we implicitly suggest an area where new reverse engineering tools might be helpful — namely, finding such trouble spots. The ability to do this will not change the underlying intractability but might incrementally help those stuck with reverse engineering.

3. The Nature of the Reverse Engineering Task

At first glance, the conclusion that reverse engineering of legacy code is doomed to fail strikes most people as either ridiculous and wrong (the "reverse engineering advocates" camp), or obvious and trivial (the "reverse engineering skeptics" camp). Some hedge, claiming it could be either depending on the definition of reverse engineering.

By the putative definition quoted in Section 1, reverse engineering involves achieving an "understanding" [3, 12, 14] of a system, including "identifying the components of an existing software system and the relationships among them" and "creating high-level descriptions". What does this mean? We argue that successful reverse engineering of a legacy system entails at least the following two subtasks:

(1) Identifying the functional components of the system and the roles they play in producing the behavior of the higher-level system that employs them.

(2) Creating a valid explanation of *how* and *why* the behavior of the higher-level system arises from these functional components and their roles.

We use "functional" here in the sense of contributing to functional run-time behavior. This means that the relevant components of a system, from the standpoint of understanding system behavior, are not necessarily the structural components of its source code (e.g., modules, subroutines, loop bodies, statements). Some functional components might correspond to easily-identified structural components, but others might span several of them — especially where interesting behavior arises from poor design or from unanticipated interactions between structural components.

By "valid explanation" we mean, effectively, a proof that the claimed higher-level behavior results from the identified functional components and roles. The challenges in achieving understanding of a poorly understood system are to generate a hypothesis, which is in fact correct, and to establish why it is correct.

3.1. Testing vs. Proving

We now consider the claim, "People do reverse engineering all the time; how can it be prohibitively expensive?" Certainly one can define reverse engineering so this is true. But what people really do all the time is to make plausible hypotheses. They do not check the validity of those hypotheses in any decisive way. They might, for instance, make some changes to the code that should not cause problems according to the hypothesis, then test to see whether those changes cause obvious problems.

Such an approach can only hope to show that a hypothesis is invalid, not that it is valid — a conclusion similar to the well-known aphorism that program testing can only hope to demonstrate the presence of bugs, not their absence. We do not trick ourselves into believing we have built correct software by *defining the problem* of building correct software in such a way that testing alone is sufficient to decide whether we have succeeded. Yet defining reverse engineering to consist of hypothesize-and-*test*, not hypothesize-and-*prove*, amounts to the same thing.

Advocates of the weaker definition might contend that all they are hoping for is to obtain "approximate" understanding of a system. But an approximation is *not* sufficient to achieve the ultimate objective of reverse engineering. Furthermore, we can find no reasonable technical definition of "approximate" reverse engineering. In any event there must be an absolute standard by which to judge the quality of an approximation. We therefore define successful reverse engineering to entail decisive checking of the validity of hypotheses, not merely guessing.

3.2. Substantive Hypotheses

The reverse engineering hypothesis should contribute enough to the understanding of a system to suggest and/or rule out potential modifications that are intended to achieve the objective of the project. Biggerstaff, *et al.*, seem to summarize nicely:

> A person understands a program when able to explain the program, its structure, its behavior, its effects on its operational context, and its relationships to its application domain in terms that are qualitatively different from the tokens used to construct the source code for the program. [2, p. 72]

We therefore stipulate that reverse engineering hypothesis H for system S should be *substantive* in that it is:

- *Effective* — it provides the ability to predict relevant behavior of S (e.g., relevant input-output behavior) and to answer questions about what-if situations (e.g., the effects of various changes to the source code of S).
- *Comprehensive* — its validity cannot be decided by a small set of test cases.
- *Concise* — it is at worst not much bigger than the source code of S.
- *Independent* — it is not a paraphrase of the code of S.
- *Systemic* — judging its validity requires examining essentially all the code of S.

The first property is basic to utility. All the others are technically necessary to rule out trivial hypotheses that might otherwise be seen as counterexamples to intractability, but which in practice contribute nothing to the understanding of S. These conditions are not really very strong. For example, nearly every non-trivial hypothesis is systemic because there are many ways to get S to exhibit unhypothesized behavior via long-range weird interactions among its components. Whether a particular system actually has such interactions does not even matter; they *might* exist because they are not ruled out by static (e.g., programming language) constraints. An instruction that influences whether and why H holds might be lurking anywhere in the code of S, and there is simply no way to know whether it is there without looking for it.

4. The Intractability Result

The particular computational problem that we claim to be intractable is the second reverse engineering subtask:

> **EXPLAIN** — Given as input (S, H) — source code for a system S and hypothesis H about that system's behavior — decide whether, and explain why, H does or does not hold for S.

We do not need to account for the extra time it takes to generate a hypothesis to be explained. There is every reason to suspect that generating substantive hypotheses is hard, too, but we do not need to or try to demonstrate this.

We claim there is a lower bound for EXPLAIN for valid hypotheses which implies that reverse engineering of legacy code (as defined in Section 3) is intractable:

EXPLAIN is Intractable — There is a constant c>1 such that, for every legacy system S and every valid substantive hypothesis H, EXPLAIN(S,H) takes time at least $c^{|S|}$, where |S| is the size of S's source code.

This result follows from two premises, which we outline here because they are empirical statements which, in principle, are falsifiable and therefore debatable. The main argument is completed elsewhere [19].

4.1. Source Code is a Compact Representation of Behavior

It has long been accepted by software and other engineers that the key to dealing with large systems is to design and construct them by composing some smaller units that are independent except at their interfaces — the objective of *modularity*. One intended result of modularity is the ability to reason modularly about program behavior. Liskov and Guttag clearly state this objective in their description of how we should like to reason about total correctness, but the conclusion applies equally to reasoning about any substantive hypothesis about system behavior:

> We reason separately about the correctness of a procedure's implementation and about parts of the program that call the procedure. To prove the correctness of a procedure definition, we show that the procedure's body satisfies its specification. When reasoning about invocations of a procedure, we use only the specification. [11, p. 227-228]

This observation is based on something routinely taught to first-year programmers: It is hopeless to reason about execution of non-trivial programs by tracing instruction execution sequences, either for particular values or by symbolic execution, because even a small program can describe arbitrarily long execution sequences through recursive calls and looping. (Effective reasoning about program behavior also requires loop bodies to be replaced by specifications, e.g., loop invariants or loop functions.) In short, it must be possible to reason about the effect of any repeatedly-executed piece of code by using a specification of that piece, *without tracing the code* for each dynamically-occurring use of it. We take as a premise that software engineers strive to achieve, and succeed in achieving, part of what they have been taught — to encode long execution sequences in a concise way by identifying commonalities in source code and by factoring them out into separate pieces that are used repeatedly.

Consider any instruction execution sequence E of system S, and define |E| as the length of a record of the steps (say, instructions) taken in E. We claim:

> **Compact Source Code Premise** — There is a constant $c > 1$ such that, for every legacy system S and for every substantive hypothesis H, there is some instruction execution sequence E which H purports to explain and for which $|E| > c^{|S|}$.

This premise is really quite a weak statement about legacy systems because most real code describes potential execution sequences that are not bounded *a priori* by any function of |S|, but only by the inputs to S. Consider that if E were achieved by straight-line code, for example, then we would need to have $|S| \geq |E|$. How could this hold for any realistic system? Rephrased in these terms, the premise says the source code for a real legacy system is substantially smaller than the length of the longest behavior history it can effect, i.e., its size is at most $\log_c |E|$. Clearly this always holds where there is no *a priori* bound on the longest execution sequence of S.

4.2. Problems Result From Failed Attempts at Modularity

We should hope that software engineers always succeed in separating specification from implementation in a way that achieves modularity. However, designing and implementing code that supports modular reasoning about behavior is more subtle than it appears at first [13, 20]. Problems arise from coupling through side-effects and aliased variables [4, 7], arrays, pointers, and dynamic storage management [6, 8], generics [5], inheritance [10, 16], and from many other sources. Potentially troublesome techniques are permitted by the programming languages used for real legacy systems because, in the interest of performance and other essential considerations, these techniques can be useful when applied carefully.

However, history gives no evidence that software engineers in practice do — or that they even know how to — exercise adequate care in the use of such powerful language constructs. We therefore claim:

> **Non-Modularity Premise** — Every legacy system is hard to maintain because, in some crucial places, it has been designed or coded so that modularity is not achieved.

We need make no assumption about how the legacy system got into this state. Perhaps the system was poorly understood from day one, or perhaps became poorly understood through the cumulative toll of patches, upgrades, and adaptations. Whatever the cause, when an "existing" system graduates to the status of "legacy" system it has already been observed to be difficult to maintain. Non-modularity of reasoning about its behavior is a major reason for this.

5. Conclusion

Reverse engineering of large legacy systems is intractable in the following sense: Given real legacy code, the time required to show the validity of a proposed explanation for why it exhibits any significant system-level behavior is at least exponential in the size of the source code. This does not mean that the task is impossible. It means that it is prohibitively costly for large legacy systems.

One lesson from this should be that we need to put more emphasis, not less, on careful engineering of new systems [13]; and that this emphasis needs to focus (at least) on creating systems that admit modular reasoning. There are many good reasons to continue to work on reverse engineering of legacy code — it is an exciting intellectual

challenge and a problem that sometimes has to be faced in practice. But at the same time we need to be realistic about what outcomes to expect. Researchers and developers, and especially their sponsors and the customers buying their wares, should not be disappointed that nothing seems to work very well for large legacy systems.

Acknowledgment

Dean Allemang, B. Chandrasekaran, Steve Edwards, John Hartman, Doug Kerr, Tim Long, Bill Ogden, Murali Sitaraman, Neelam Soundararajan, Michael Stovsky, Sergey Zhupanov, and Stu Zweben have provided helpful insights and/or feedback on the ideas presented here. We also gratefully acknowledge the support of the National Science Foundation through grant CCR-9311702; and the Advanced Research Projects Agency under ARPA contract number F30602-93-C-0243, monitored by the USAF Materiel Command, Rome Laboratories, ARPA order number A714.

Bibliography

[0] Andersen, H.C. *The Emperor's New Clothes: A Fairy Tale*, Addison-Wesley, Reading, MA, 1973.

[1] Bennett, K. Legacy systems: coping with success. *IEEE Software 12*, 1 (Jan. 1995), 19-23.

[2] Biggerstaff, T.J., Mitbander, B.G., and Webster, D.E. Program understanding and the concept assignment problem. *Comm. ACM 37*, 5 (May 1994), 72-83.

[3] Chandrasekaran, B., Goel, A.K., and Iwasaki, Y. Functional representation as design rationale. *Computer 26*, 1 (Jan. 1993), 48-56.

[4] Cook, S.A. Soundness and completeness of an axiom system for program verification. *SIAM J. Comp. 7*, 1 (Feb. 1978), 70-90.

[5] Ernst, G.W., Hookway, R.J., Menegay, J.A., and Ogden, W.F. Modular verification of Ada generics. *Comp. Lang. 16*, 3/4 (1991), 259-280.

[6] Ernst, G.W., Hookway, R.J., and Ogden, W.F. Modular verification of data abstractions with shared realizations. *IEEE Trans. on Software Eng. 20*, 4 (Apr. 1991), 288-307.

[7] Harms, D.E., and Weide, B.W. Copying and swapping: influences on the design of reusable software components. *IEEE Trans. on Software Eng. 17*, 5 (May 1991), 424-435.

[8] Hollingsworth, J.E. *Software Component Design-for-Reuse: A Language-Independent Discipline Applied to Ada.* Ph.D. dissertation, Dept. of Comp. and Inf. Sci., Ohio State Univ., Columbus, OH, Aug. 1992; available from "ftp.cis.ohio-state.edu" in "/pub/tech-report/1993/TR01-DIR/*".

[9] Hopkins, J.E., and Sitaraman, M. Software quality is inversely proportional to potential local verification effort. *Proc. 6th Ann. Workshop on Software Reuse*, Owego, NY, Nov. 1993.

[10] Leavens, G.T., and Weihl, W.E. Reasoning about object-oriented programs that use subtypes. *Proc. OOPSLA '90/SIGPLAN Notices 25*, 10 (Oct. 1990), 212-223.

[11] Liskov, B., and Guttag, J. *Abstraction and Specification in Program Development.* McGraw-Hill, New York, 1986.

[12] Littman, D.C., Pinto, J., Letovsky, S., and Soloway, E. Mental models and software maintenance. In *Empirical Studies of Programmers*, E. Soloway and S. Iyengar, eds., Ablex, 1986, 80-98.

[13] Neumann, P.G. Are dependable systems feasible? *Comm. ACM 36*, 2 (Feb. 1993), 146.

[14] Parnas, D.L., Madey, J., and Iglewski, M. Precise documentation of well-structured programs. *IEEE Trans. on Software Eng. 20*, 12 (Dec. 1994), 948-976.

[15] Waters, R. C., and Chikovsky, E. Reverse engineering progress along many dimensions. *Comm. ACM 37*, 5 (May 1994), 23-24.

[16] Weber, F. Getting class correctness and system correctness equivalent: how to get covariance right. In *Proc. TOOLS USA '92*, R. Ege, M. Singh, and B. Meyer, eds., Prentice-Hall, 1992.

[17] Weide, B.W., Heym, W.D., and Ogden, W.F. Procedure calls and local certifiability of component correctness. *Proc. 6th Ann. Workshop on Software Reuse*, Owego, NY, Nov. 1993.

[18] Weide, B.W., and Hollingsworth, J.E., *On Local Certfiability of Software Components*, tech. report OSU-CISRC-1/94-TR04, Dept. of Comp. and Inf. Sci., Ohio State Univ., Columbus, OH, Jan. 1994; available from "ftp.cis.ohio-state.edu" in "/pub/tech-report/1994/TR01.ps.gz".

[19] Weide, B.W., Heym, W.D., and Hollingsworth, J.E., *Reverse Engineering of Legacy Code is Intractable*, tech. report OSU-CISRC-10/94-TR55, Dept. of Comp. and Inf. Sci., Ohio State Univ., Columbus, OH, Oct. 1994; available from "ftp.cis.ohio-state.edu" in "/pub/tech-report/1994/TR55.ps.gz".

[20] Wilde, N., Matthews, P., and Huitt, R. Maintaining object-oriented software. *IEEE Software 10*, 1 (Jan. 1993), 75-80.

Modeling Modular Software Structure for Human Understanding

Stephen H. Edwards
Dept. of Computer and Information Science
The Ohio State University
2015 Neil Avenue
Columbus, Ohio 43210–1277
E-Mail: edwards@cis.ohio-state.edu
URL: http://www.cis.ohio-state.edu/~edwards

Abstract

People form internal mental models of the things they interact with in order to understand those interactions. This psychological insight has been used by the human-computer interaction (HCI) community to build software systems that are more intuitive for end users, but it has only been informally applied to the problems of software designers, programmers, and maintainers. Conventional programming languages still do little to help client programmers develop good mental models of software subsystems.

To address this problem, we have developed the Abstract and Concrete Templates and Instances (ACTI) model of modular, parameterized software subsystems. This model of software structure addresses the needs of human software engineers who must reason about collections of interacting software parts during design, maintenance, and evolution.

ACTI is different from other module systems and models of software in several ways. In ACTI, a subsystem never *has any implicit dependencies, and never depends directly on any external definitions—all external dependencies are described through an explicit interface. In addition, a subsystem specification is meaningful by itself, even without respect to any implementation. Finally, a subsystem is more than just a collection of types and operations; it also includes: an explicit model of behavior, an explicit model of all external dependencies, a collection of definitions used to construct and describe these models, and (potentially complex) substructure. There are strong parallels between ACTI and other research on the understanding of modularly structured physical devices, particularly Functional Representation.*

Keywords: *Mental model, model-based specification, interfaces, bindings, generics*

1 Introduction

Modern programming languages have evolved from their predecessors with the primary purpose of describing instructions to computers. Generally, these languages were not designed to help explain to people the meaning of the software that they can describe. This has led to two significant problems with programming languages today: modules are considered to be purely syntactic constructs with no independent meaning, and those parts of programs that are deemed meaningful (usually procedures, in imperative languages) have "hierarchically constructed" meanings.

To address these deficiencies, here we outline a new model of component-based software that provides concrete support for recording critical information about each software structure, information that can form the basis for a programmer's own mental model of that structure. This new model, termed the ACTI model (for "Abstract and Concrete Templates and Instances") is both mathematically formal and programming language-independent. It captures and formalizes the underlying conceptual view of software architecture embedded in modern module-structured languages while simultaneously providing support for forming mental models. As a result, it can serve as a general-purpose theory of the nature of software building-blocks and their compositions.

1.1 Why Conventional Languages Fail

Most modern programming languages have some construct that is intended to be the primary "building-block" of complex programs. This building-block may be called a "module," a "package," a "structure," or a "class." Unfortunately, these constructs are rarely given meaningful semantic denotations. Conventional wisdom in the computer science field is that these constructs are primarily for grouping related defini-

tions, controlling visibility, and enforcing information hiding. For example, when considering module-structured languages like Ada or Modula-2, Bertrand Meyer writes:

> In such languages, the module is purely a syntactic construct, used to group logically related program elements; but it is not itself a meaningful program element, such as a type, a variable or a procedure, with its own semantic denotation. [1, p. 61]

In this view, there is no way for one to make such building-blocks contribute directly to the understandability of the software comprising them. While object-oriented languages usually give a stronger intuitive meaning to the notion of a "class," they also fail to provide any vision of how the meaning of individual classes can contribute to a broader understanding of the software systems in which they are embedded.

In addition, those program elements that *are* given a semantic denotation are often given a meaning that is "hierarchically constructed," or synthesized. In other words, the meaning of a particular program construct, say a procedure, is defined directly in terms of its implementation—a procedure "means" what the sequence of statements implementing it "means." The meaning of its implementation is defined in terms of the meanings of the lower-level procedures that it calls. Thus, a procedure's meaning is constructed from the meanings of the lower-level program units it depends on, and the meanings of those lower-level units in turn depend on how they are implemented, and so on.

This simple synthesis notion of how meaning is defined bottom-up is adequate from a purely technical perspective. It is also very effective when it comes to describing the semantics of layered programming constructs. Unfortunately, it is at odds with the way human beings form mental representations of the meanings of software parts [2].

The result of these two features of existing programming languages is that they are inadequate for effectively communicating the meaning of a software building-block to people (programmers, in particular). The semantic denotations of programming constructs in current languages only relate to *how* a program operates. They fail to capture *what* a program is intended to do at an abstract level, or *why* the given implementation exhibits that particular abstract behavior. In order to address these concerns, it is necessary to assign meaning to software building-blocks, to separate the abstract description of a software part's intended behavior from its implementation, and to provide a mechanism for explaining why the implementation of the part achieves behavior consistent with that abstract description.

1.2 Toward Understandable Software

The ACTI model directly addresses these deficiencies of current programming languages by giving a software subsystem a well-defined meaning of its own, independent of how it may be implemented. This meaning includes an explicit model of behavior, which can serve as a reference to help a client programmer understand the subsystem—form an effective mental model of it. Further, the constant presence of such a behavioral description acts as a continual cue to aid the programmer in maintaining the consistency and correctness of her own understanding.

Because a person's internal mental representations are so critical for comprehension, supporting the formation and maintenance of effective mental models is important if one wishes to support complex software structures that are understandable by humans. Understandable software is vital for software designers, who must design in the context of reusable software parts; for testing and maintenance personnel, who often must understand software written by others; and for reverse engineers or re-engineers, who want to gain as much value from previous work as possible.

Although a full treatment of ACTI's formal definition is beyond the scope of this article because of space considerations, the following sections provide a general overview of the model at an intuitive level. Section 2 introduces the main entities in ACTI. Next, Section 3 highlights the unique and novel features of the model. Section 4 then outlines how ACTI provides support for software understanding. Relationships to previous work, particularly AI-based work on the understanding of physical devices, is discussed in Section 5.

2 An Overview of ACTI

The ACTI model [2] is centered around the notion of a "software subsystem," a generalization of the idea of a module or a class that serves as the building-block from which software is constructed. A subsystem can vary in grain size from a single module up to a large scale generic architecture. ACTI is designed specifically to capture the larger meaning of a software subsystem in a way that contributes to human understanding, not just the information necessary to create a computer-based implementation of its behavior.

The ACTI model is based on four different kinds of subsystems:

Abstract Instance—A disembodied subsystem specification or interface description. There is *no*

Table 1: The Four Kinds of Subsystems

Subsystem Varieties		
	Abstract (Specification)	Concrete (Implementation)
Template	RESOLVE concept SML functor signature	RESOLVE realization SML functor
Instance	Ada package spec. SML signature Eiffel class interface	Ada package body SML structure Eiffel class impl.

implementation associated with anything defined in the specification.

Concrete Instance—A subsystem that provides implementations for its types and operations. All of the defined types and operations in the subsystem are represented and/or implemented.

Abstract Template—A subsystem-to-subsystem function that, when applied to its argument, which is some abstract instance, will generate another abstract instance. Effectively, an abstract template is a form of generic subsystem specification.

Concrete Template—A subsystem-to-subsystem function that, when applied to its argument, which is some concrete instance, will generate another concrete instance. Thus, a concrete template is a form of generic subsystem implementation.

The terms used for this classification are all based on the work of Weide *et al.* [3, p. 23], and the same ideas appear in the 3C model [4]. The name "ACTI" is an acronym derived from these four terms: "Abstract and Concrete Templates and Instances."

This view of the world allows software subsystems to be partitioned along two orthogonal dimensions, as shown in Table 1. The distinction between "abstract" and "concrete" embodies the separation between a specification or interface, and an implementation or representation. The distinction between "template" and "instance" allows one to talk about both generic subsystems, and the product of fixing (binding) the parameters of such a generic subsystem: an instance subsystem.

Formally, ACTI is a collection of mathematical spaces, together with relations and functions on those spaces, that can be used in explaining (or defining) the denotational semantics of program constructs. In spirit, the model was developed in accordance with the denotational philosophy, as described by E. Robinson:

> In the denotational philosophy inspired by Strachey the program, or program fragment, is first given a semantics as an element of some abstract mathematical object, generally a partially ordered set, the semantics of the program being a function of the semantics of its constituent parts; properties of the program are then deduced from a study of the mathematical object in which the semantics lives. [5, p. 238]

ACTI is not a programming language, however. Instead, it is a mathematical model that is useful for programming language designers, or researchers studying the semantics of programming languages. It is a formal, theoretical model of the structure and meaning of software subsystems. It is rich enough to be used as the denotational semantic modeling space when designing new languages, and has been shown to subsume the run-time semantic spaces of several existing languages chosen to be representative of the modern imperative, OO, and functional philosophies [2].

ACTI has two features that specifically address the inadequacies described in the introduction:

1. In ACTI, a software subsystem (building-block) has an intrinsic meaning; it is not just a syntactic construct used for grouping declarations and controlling visibility. This meaning encompasses an abstract behavioral description of all the visible entities within a subsystem.

2. The meaning of a software subsystem is *not*, in general, hierarchically constructed. In fact, it is completely independent of all the alternative implementations of the subsystem.

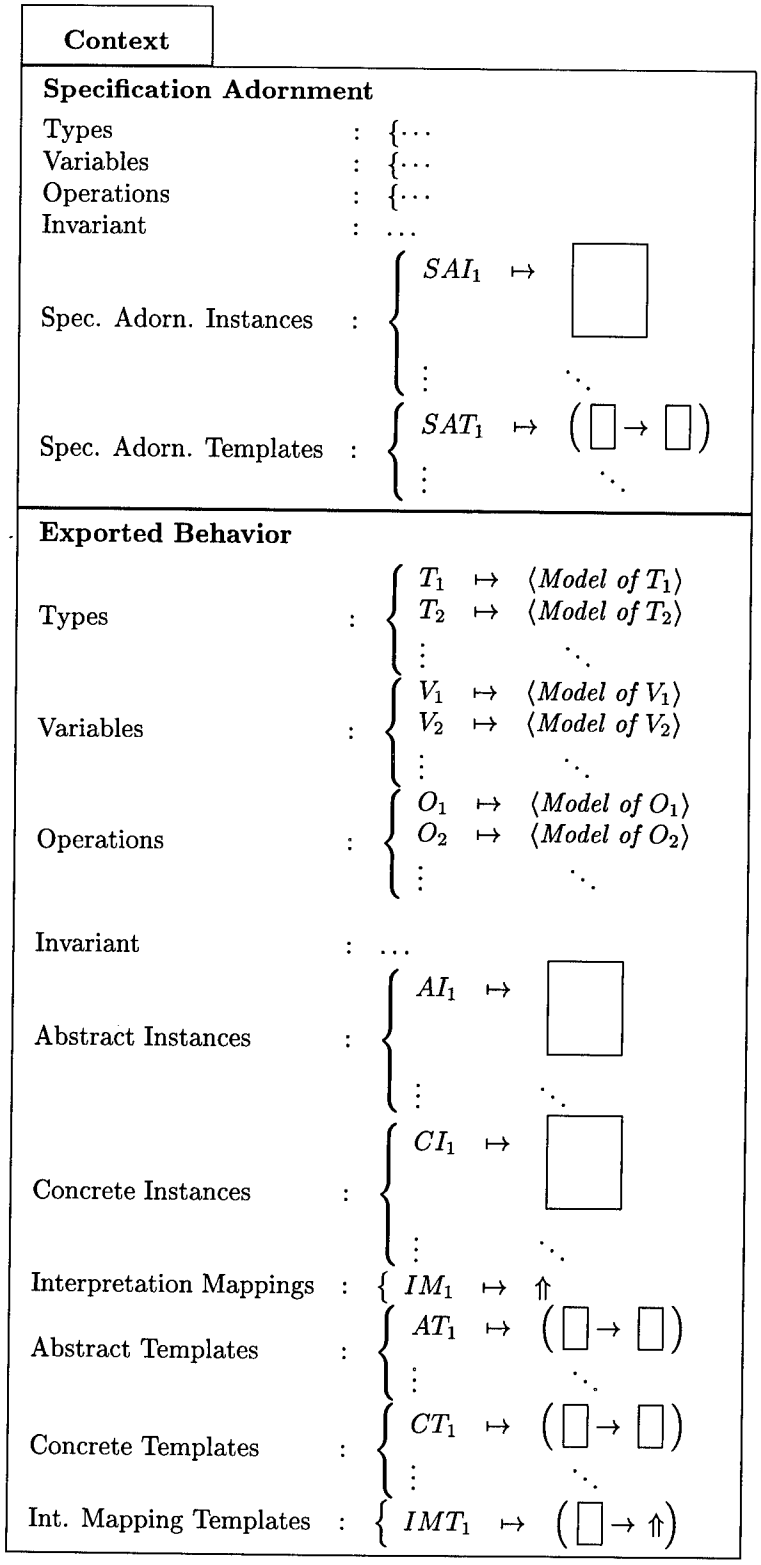


Figure 1: The Details of an Abstract Instance

Thus, ACTI provides a mechanism for describing *what* a subsystem does, not just how it is implemented. The meaning provided for a subsystem is a true abstraction—a "cover story" that describes behavior at a level appropriate for human understanding without explaining how the subsystem is implemented. Further, ACTI provides a formally defined mechanism, called an interpretation mapping, that captures the explanation of *why* an implementation of a subsystem will give rise to the more abstractly described behavior that comprises the meaning attributed to the subsystem—in short, an explanation for why the cover story works.

2.1 Abstract Instances

Table 1 gives examples of some programming language structures that might typify each of the four kinds of subsystems. We begin with abstract instances.

An abstract instance is a subsystem specification or interface description. There is *no* implementation associated with anything defined in the abstract instance. Further, like all other ACTI subsystems, an abstract instance cannot directly refer to any entities outside of itself—it is completely self-contained. If a given abstract instance relies on external definitions, they must be imported through an explicit interface that expresses exactly what expectations the instance places on its environment—an explicit "context" interface.

To briefly give the flavor of the mathematical spaces in ACTI, Figure 1 schematically depicts an abstract instance object. The abstract instance is divided into three parts, the most familiar of which is the **Exported Behavior**. The exported behavior portion of the abstract instance defines all of the services provided by the instance:

- All types that it provides, including a mathematical model space for each;
- All variables, including their types;
- All operations, including a pre- and postcondition-oriented model of their behaviors;
- An invariant over the entire instance;
- Nested abstract instances;
- (Specifications of) nested concrete instances;
- Nested interpretation mappings (interpretation mappings are described below); and
- Nested templates (templates are described below).

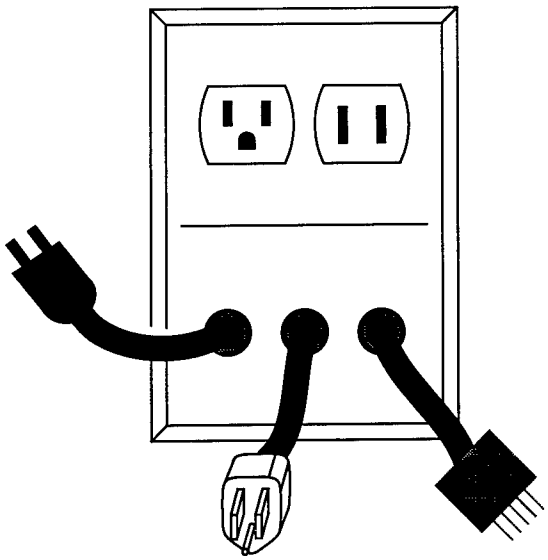


Figure 2: An Abstract Instance Is A "Face Plate"

All of these components have values taken from some complete partial order (CPO) space defined in the ACTI model.

In addition to providing a behavioral model of all exported features, ACTI includes complete behavioral descriptions of all imported features [2]. The **Context** section shown in Figure 1 is actually one (nested, possibly empty) abstract instance that is used to completely define all of the external dependencies of the main instance shown in the figure.

The remaining section of the abstract instance, the **Specification Adornment** section, is rarely manifested in programming languages. It is an area where purely mathematical definitions of types, operations, or other entities can be made, purely for use as tools in creating more understandable behavioral descriptions. While the **Exported Behavior** describes what we would normally consider to be programming-level properties of a subsystem, and while **Context** describes programming-level external dependencies, **Specification Adornment** describes mathematical specification tools.

Intuitively, we can think of an abstract instance as a "face plate" that describes an explicit interface at both the syntactic and behavioral levels, as shown in Figure 2. Here, the "sockets" on the upper half of the face plate symbolize the explicit interface to external dependencies, while the "plugs" on the lower half symbolize the features provided by this subsystem.

In ACTI, abstract instances can have

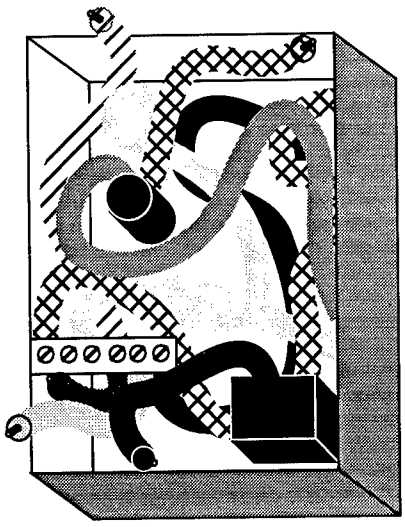


Figure 3: A Concrete Instance Is An "Open Box"

substructure—that is, abstract instances can be nested within other abstract instances. This provides a mechanism for describing cohesive groups of related features within one "face plate" as a unit, as well as for modularizing complex specifications.

2.2 Concrete Instances

A concrete instance is a subsystem that provides implementations for all of its exported features—types, operations, etc. This is much closer to the usual programming language notion of a "module." Just as with abstract instances, a concrete instance has a meaning in isolation, and cannot implicitly depend on any external entities—all external dependencies must be explicitly described in its context interface. The behavior of all features is also included in the meaning of the concrete instance. Just as with abstract instances, concrete instances can have substructure, or be nested within other concrete instances.

Unlike most programming languages, ACTI imposes no predetermined relationships between abstract and concrete instances. Implementations are meaningful in their own right (and in isolation), even without respect to any particular specification to which they may conform. While an abstract instance is in essence an "implementation-free" subsystem specification, a concrete instance is a "specification-free" implementation. The traditional notion of "conformance" between an implementation and a specification is thus many-to-many in this model.

In Figure 3, a concrete instance is shown as an electrical junction box without a face plate. Just like

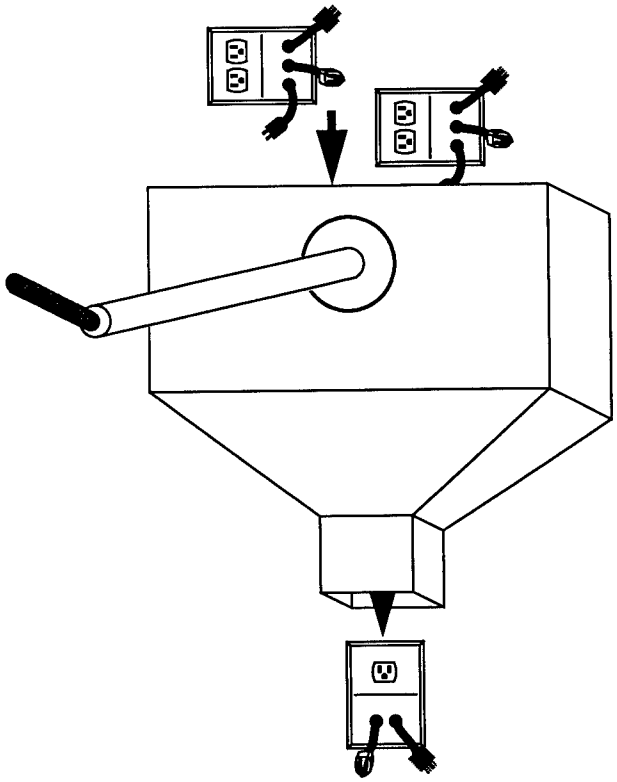


Figure 4: A Template Is A "Generator"

the abstract instance, the concrete instance has exported features (loose wires) and an explicit interface to its environment (a terminal block). Of course, one might expect the behavior of these features to be described in terms at a different level of abstraction from those used in the specification(s) to which this concrete instance conforms.

2.3 Templates

Templates in ACTI are subsystem generators. One can think of a template as a "function" that takes a subsystem as a parameter and produces a new subsystem as its result. An abstract template is a subsystem-to-subsystem function that can be applied to an abstract instance to generate another abstract instance. Effectively, an abstract template is a generic subsystem specification. A concrete template is a subsystem-to-subsystem function that can be applied to a concrete instance to generate another concrete instance. Thus, a concrete template is a generic subsystem implementation.

Figure 4 gives an intuitive impression of an abstract template. One or more abstract instances are provided as actual parameters, and the template is

applied to them (by turning the crank) to generate a new abstract instance.

2.4 Interpretation Mappings: Relationships Between Subsystems

In ACTI, the relationships between different subsystems are expressed explicitly via *interpretation mappings*. Intuitively, one can think of an interpretation mapping as being an "impedance matcher" between different abstract models of behavior. An interpretation mapping explains how one set of features can be "interpreted as" or "mapped into" another set of features in a behaviorally consistent way. This is shown in Figure 5 as a cable with differently shaped plugs. As with the other ACTI entities, there are also interpretation mapping templates for describing parameterized families of mappings between families of subsystems.

Interpretation mappings are used for several purposes:

- To explain how one abstract instance conforms to another (how one specification is a generalization of another).
- To explain how a concrete instance conforms to an abstract instance (how an implementation fulfills a specification).
- To explain how one or more external subsystems conform to the explicit context interface of an abstract or concrete instance.

Interpretation mappings are at heart explicit representations of *bindings* between subsystems. Because an ACTI subsystem intentionally includes a detailed description of its behavior, however, such a binding necessarily involves more than just a name-to-name correspondence—it must also include the equivalent of an *abstraction function* (or, more generally, *abstraction relation*) in order to bridge the gap between descriptions presented in completely different abstract terms.

While this paper can only give an overview of the concepts involved in ACTI, a strong intuitive grasp of the abstract versus concrete and template versus instance distinctions gives one an effective understanding of the heart of ACTI.

3 What Is Different Here?

All of the varieties of subsystems modeled in ACTI have already appeared in modern programming languages in one form or another—although rarely do all four appear together, and there is much disagreement about their details. The contributions of ACTI and its crucial differences are in the details of subsystems and how they fit together.

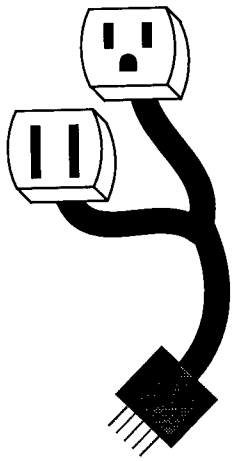


Figure 5: Interpretation Mappings Are Connectors

In ACTI, subsystems are meaningful by themselves. In particular, a specification has a well-defined meaning, including a complete picture of the behavior of the features it describes, even without respect to any implementation. An implementation also has meaning, without reference to any specification to which it might conform.

As a result, subsystems never depend directly on anything outside of themselves. All external dependencies are described through an explicit context interface, and there is no notion of "implicit" dependencies or hidden coupling between subsystems.

The "meaning" of a subsystem (a specification or implementation) is more than just a collection of types and operations. This is critically different than the notion of "module" in most programming languages. Instead, the meaning of a subsystem includes:

- An explicit model of behavior (independent of implementation details).
- An explicit model of all external dependencies (the context interface).
- A collection of definitions used to construct and describe behavioral models.
- Substructure (which is potentially complex).

In addition to the emphasis on subsystems, ACTI also includes explicit representation of correspondence

relationships between subsystems. In ACTI, all *bindings* are explicitly represented through these interpretation mappings:

- Binding an external unit to a subsystem's explicit context interface.
- Mapping a concrete instance to the abstract instance(s) it conforms to.
- Mapping an abstract instance to another abstract instance it conforms to.

Finally, in ACTI all entities can be parameterized. Implementations can be parameterized independently of specifications, and interpretation mappings can be parameterized independently of the subsystems they relate. Examples of the utility of this flexibility are surprisingly plentiful [2].

4 Support For Software Understanding

It is well-accepted that people form *mental models*—internal representations of external artifacts—for devices and other bits of technology with which they interact [6, p. 241]. From psychology, we understand that people do this naturally, and that such models help individuals in two ways [6, p. 241]:

1. A mental model allows one to *predict* the behavior of the person or thing with which the interaction takes place.
2. A mental model allows one to *explain why* the behavior arises.

Both of these benefits are important for a person to understand how to interact effectively with another person, a physical device, or a piece of complex software. Hence, a mental model that is *effective* is one that provides sufficient predictive and explanatory power, and which a person can reasonably internalize and use to understand an interaction.

Here, we are concerned with a "programmer-user's" interactions with a software subsystem, rather than with an end-user's interactions with a complete application. Following Norman's terminology [6], *target system* will be used to refer to the component subsystem with which a person is interacting. The *system image* is the entire visible "programmer interface" to the software component seen by a(nother) software professional. It may include a system specification, complete source code, manuals and instructions accompanying the software, and even the way the software behaves and responds under operating conditions.

Mental models evolve naturally through interaction with the target system [6, p. 241]. Over time, people reformulate, modify, and adapt their mental models whenever these models fail to provide reasonable predictive or explanatory power. For most purposes, the models need not be completely accurate, and usually they are not, but they must be functional. Norman documents the following general observations about mental models [6, p. 241]:

1. Mental models are incomplete.
2. People's abilities to simulate or mentally execute their models are severely limited.
3. Mental models are unstable: People forget the details of the system they are using, especially when those details (or the whole system) have not been used for some period.
4. Mental models do not have firm boundaries: similar devices and operations get confused with one another.
5. Mental models are "unscientific": People maintain "superstitious" behavior patterns even when they know they are unneeded because they cost little in physical effort and save mental effort.
6. Mental models are parsimonious: Often people do extra physical operations rather than the mental planning that would allow them to avoid those actions.

These observations indicate that mental models are inherently limited. These limitations stem from human cognitive limitations, a person's previous experiences with similar systems, and even misleading system images [6, p. 241]. As Norman points out:

> In making things visible [in the system image], it is important to make the correct things visible. Otherwise people form explanations for the things they can see, explanations that are likely to be false. ... People are very good at forming explanations, at creating mental models. It is the designer's task to make sure that they form the correct interpretations, the correct mental models: the system image plays the key role. [7, p. 198]

Given this information, how can one support the formation and maintenance of effective mental models for complex software systems? Norman points out

that the designer of a software subsystem already has a *conceptual model* of the system that is (presumably) accurate, consistent, and complete [6, p. 241][7, pp. 189-190]. The goal, in the best of all possible worlds, is to ensure that the system image is completely consistent with the conceptual model of the designer, and that from this system image the user forms a mental model consistent with the designer's conceptual model. Further, the system image should help alleviate the inherent human limitations of mental models. A well-designed system image can help to make the user's mental model more complete, to record details so the user's model has firm boundaries, and to present those details in such a way that the user's model is more stable.

ACTI was designed with these issues in mind, and as a result, explicitly incorporates model-based explanations of behavior in the meaning of each software subsystem. This behavioral description captures what the component designer wishes to impart about the conceptual model he intends for the client software engineer to acquire—it is a semantic (rather than purely syntactic) system image.

Because an ACTI subsystem has a meaning that contains the designer's conceptual model, it can serve as a cue to help the programmer *form and maintain* a mental model of the subsystem. By serving as a point of reference, it also helps to address the natural limitations of the programmer's internal model. This is the basis for supporting software that is understandable by humans, not just executable by machines.

5 Relation to Previous Work

In the realm of computer languages, Section 3 delineates the primary ways in which ACTI is unique with respect to previous work. For a discussion of previous efforts to effectively capture modular structuring techniques, Edwards [2] presents a thorough comparison of several programming languages that are representative of best current practices: RESOLVE [8, 9], OBJ [10], Standard ML [11], and Eiffel [1]. As typical of current efforts, most of these languages inadequately support the formation or maintenance of effective mental models of software parts. The complete analysis, including a detailed check list of software structuring and composition properties supported by these languages, is available electronically [2]. Other efforts to provide better support for mental models have concentrated primarily on adding (possibly structured) comments to component specifications, the inadequacy of which is explained in [12].

Interestingly, there are other areas of computer science where similar work has been and is being carried out. Current work on Functional Representation [13, 14, 15, 16] (FR) is closely related. FR grew out of artificial intelligence work on reasoning about physical systems, partly motivated by diagnostic and design problem solving. As with other AI work on these problems, FR originally focused on the functions of devices—that is, the effects objects have on their environment. More recently, B. Chandrasekaran has documented an ontological framework within which the notion of "function" can be explained, and which provides a unified technical vision underlying the various approaches to device understanding [17].

This more general framework has allowed Chandrasekaran to present FR as a general theory of comprehension, along with a specific language that serves as a corresponding representation mechanism [17]. He defines comprehension to be the task of producing one or more of the following descriptions of a given artifact:

- The intended **function** of the artifact, i.e., its behavior.
- The **structure** of the artifact, i.e., a specification of its components and how they are put together.
- A **causal account** of how the artifact achieves its function, and the roles played by the components in achieving it.

The result is that FR can be considered to be a domain-independent theory for understanding how system-level behaviors emerge from a system's structure.

ACTI is aimed at exactly the same goal within the domain of software systems. As a result, it is not surprising that ACTI and FR share a number of critical features:

- Support for human understanding is a primary goal.
- Behavior is described independently of structure and implementation. In particular, behavioral descriptions are meaningful in isolation, without respect to any particular device that may provide such behavior.
- There is the potential for multiple realizations of the same behavior.
- An explicit bridge must be constructed between the (more abstract) vocabulary of a system-level behavioral description and the (lower level) functions achieved by the system's component parts.

- Similarly, when devices are composed to form larger structures, one must explicitly describe the bridge between them.

- The "context interface" through which external forces can act on a given artifact is explicitly defined.

Earlier FR work has even been applied to software for diagnostic and explanation purposes [18, 19]. To date, this has been at the "programming-in-the-small" level of individual statements and their interactions, while ACTI addresses "programming-in-the-large" issues of subsystem meaning and composition.

The similarities between ACTI and FR, two efforts that were arrived at independently, strengthen the claim that they both make progress toward supporting human understanding effectively. Further, there are areas where the two complement each other. FR, for example, provides for explicit representation of a "causal account" of how aggregate behavior arises from the functions of individual parts, which can contribute to human understanding for modification or diagnostic purposes. Similarly, ACTI provides explicit support for specification adornments—a place for designers to capture model-building tools and useful subparts of behavioral descriptions. The complementary nature of these parallel efforts provides fruitful ground for future work on integrating the two frameworks.

6 Conclusions

ACTI addresses the problem of *understandable* software composition across module- and class-based languages. To achieve this, it gives a real semantic denotation to subsystems (modules) that includes a simple (i.e., not "bottom-up") model of behavior. It allows one to clearly describe why abstractions (simple models) correctly capture the behavior of complex combinations of lower-level parts, using interpretation mappings.

Because each ACTI subsystem has a meaning that reflects its designer's conceptual model, it supports the formation of mental models by client programmers, and also addresses their limitations. This is a critical missing link in the support of understandable software that has been ignored by previous efforts in programming language design.

ACTI is universal, in that it is not tied to any particular programming language. It unifies module-structured and object-oriented notions of software, and can serve as a general theory of software structure and meaning.

Acknowledgements

The author gratefully acknowledges financial support from the National Science Foundation (grant number CCR-9311702) and the Advanced Research Projects Agency (contract number F30602-93-C-0243, monitored by the USAF Materiel Command, Rome Laboratories, ARPA order number A714). B. Chandrasekaran deserves special thanks for recognizing and cultivating the ties between Functional Representation and ACTI.

References

[1] B. Meyer, *Object-Oriented Software Construction.* New York, NY: Prentice Hall, 1988.

[2] S. Edwards, *A Formal Model of Software Subsystems.* PhD thesis, Dept. of Computer and Information Science, The Ohio State University, Columbus, OH, 1995. Also available as technical report OSU-CISRC-4/95-TR14, by anonymous FTP from `ftp://ftp.cis.ohio-state.edu/pub/tech-report/1995/TR14-DIR`, or through the author's home page.

[3] W. F. Ogden, M. Sitaraman, B. W. Weide, and S. H. Zweben, "The RESOLVE framework and discipline—a research synopsis," *ACM SIGSOFT Software Engineering Notes*, vol. 19, pp. 23–28, Oct. 1994.

[4] W. Tracz, "Implementation working group summary," in *Reuse in Practice Workshop Summary* (J. Baldo, Jr., ed.), (Alexandria, VA), pp. 10–19, IDA Document D-754, Institute for Defense Analyses, Apr. 1990.

[5] E. Robinson, "Logical aspects of denotational semantics," in *Category Theory and Computer Science* (D. H. Pitt, A. Poigné, and D. E. Rydeheard, eds.), vol. 283 of *Lecture Notes in Computer Science*, pp. 238–253, New York, NY: Springer-Verlag, 1987.

[6] D. A. Norman, "Some observations on mental models," in *Readings In Human-Computer Interaction: A Multidisciplinary Approach* (R. M. Baecker and W. A. S. Buxton, eds.), pp. 241–244, San Mateo, CA: Morgan Kaufmann Publishers, Inc., 1987.

[7] D. A. Norman, *The Design of Everyday Things.* New York, NY: Doubleday/Currency, 1990.

[8] M. Sitaraman and B. W. Weide, editors, "Special feature: Component-based software using RESOLVE," *ACM SIGSOFT Software Engineering Notes*, vol. 19, pp. 21–67, Oct. 1994.

[9] B. W. Weide, W. F. Ogden, and S. H. Zweben, "Reusable software components," in *Advances in Computers* (M. C. Yovits, ed.), vol. 33, pp. 1–65, Academic Press, 1991.

[10] J. A. Goguen, "Principles of parameterized programming," in *Software Reusability, Volume I: Concepts and Models* (T. J. Biggerstaff and A. J. Perlis, eds.), pp. 159–225, New York, NY: ACM Press, 1989.

[11] R. Milner, M. Tofte, and R. Harper, *The Definition of Standard ML.* Cambridge, MA: MIT Press, 1990.

[12] S. H. Edwards, "Good mental models are necessary for understandable software," in *Proceedings of the Seventh Annual Workshop on Software Reuse* (L. Latour, ed.), Aug. 1995.

[13] B. Chandrasekaran, "Functional representation and causal processes," in *Advances in Computers* (M. C. Yovits, ed.), vol. 38, pp. 73–143, Academic Press, 1994.

[14] Y. Iwasaki, R. Fikes, M. Vescovi, and B. Chandrasekaran, "How things are intended to work: Capturing functional knowledge in device design," in *Proceedings of the 13th International Joint Conference on Artificial Intelligence*, pp. 1516–1522, Morgan KAufmann, 1993.

[15] M. Vescovi, Y. Iwasaki, R. Fikes, and B. Chandrasekaran, "CFRL: A language for specifying the causal functionality of engineered devices," in *Eleventh National Conference on AI*, pp. 626–633, AAAI Press/MIT Press, 1993.

[16] Y. Iwasaki and B. Chandrasekaran, "Design verification through function- and behavior-oriented representation," in *Artificial Intelligence in Design '92* (J. S. Gero, ed.), pp. 597–616, Kluwer Academic Publishers, 1992.

[17] B. Chandrasekaran, "An explication of function." The Ohio State University, Laboratory for AI Research, draft, 1996. Also available electronically from `http://www.cis.ohio-state.edu/~chandra`.

[18] D. Allemang, "Using functional models in automatic debugging," *IEEE Expert*, vol. 6, pp. 13–18, 1991.

[19] B. Liver and D. T. Allemang, "A functional representation for software reuse and design," *International Journal of Software Engineering and Knowledge Engineering*, vol. 5, pp. 227–269, 1995.

Representing Function as Effect: Assigning Functions to Objects in Context and Out

B. Chandrasekaran and John R. Josephson

Laboratory for AI Research
The Ohio State University
375, Dreese Laboratories
2015 Neil Avenue
Columbus, OH 43210
{Chandra, jj}@cis.ohio-state.edu

Abstract

Various recent representations for device function capture somewhat different intuitions and are restricted in their ranges of applicability. Though each representation solves some set of problems, it is hard to see how to build on them. In this paper, we describe a formal framework and definition of device function that attempts to unify and generalize these intuitions. We have sought the smallest ontological framework that is sufficient for developing an idea of function that will support design problem solving through the use of functionally indexed device libraries. As characterized in this ontology, objects are embedded in an environment and represented in a view. Objects interact causally with their environments. Functions are defined in terms of the effects of objects on their environments. This definition of function allows for functions to be specified as requirements during the design process, and prior to choosing objects to achieve those functions. An object represented in a device library is associated with generic environmental properties that are bound to specifics when the object is deployed. In this way potential functions are associated with library objects. In the universe of engineered objects, causal dependencies can usually be expressed as relations between the properties of objects and property relations can usually be expressed as mathematical functions. However, a formalism based on property relations appears not to be sufficient to capture all types of causality relevant to functional reasoning. This paper presents a definition of function that is sufficiently general to express static and dynamic functions, intended and natural functions, and functions of abstract and physical objects.

Desiderata for a Framework for Functions

The last two decades have seen a flowering of work on reasoning about physical systems. Recently, motivated by problems in diagnosis and design, a stream of work has emerged in which the notion of device function has been central. This stream of work uses many of the ideas that have been developed in qualitative reasoning and qualitative simulation. For example, [1] uses qualitative simulation to verify design functions.

The various investigations of device function have mostly lacked a unified technical vision. Different intuitions about functions are pursued in different contexts and application domains. Even though each approach clearly solves some set of problems, it is hard to see how unify or to build on them. What we need is a minimalist effort, something that looks for what is common among all the intuitions about function and seeks to build the smallest ontological framework within which an adequate notion of function can be explicated. Of course, for work in particular domains and applications, additional constructs and content theories will be needed, but, if the effort is successful, the minimalist ontology will be usable by all. This paper is an attempt to provide such an ontology.

A framework for functions should, in our opinion, satisfy the following desiderata.

- 1. It should apply to intended functions of human-designed devices, and to functions or roles in natural systems.
- 2. It should apply to functions of both static and dynamic objects. Almost all of the work on reasoning about objects and their functions has focused on functions that are defined in terms of state changes of objects, e.g., electronic circuits, buzzers, gears, and so on. However, the notion of function applies just as well to static objects, e.g., support beams and windows.
- 3. It should apply to functions of both abstract and physical objects. Even though most work has been done for physical objects, one can speak of functions of modules in software, and of steps in plans, just as naturally as speaking of functions of physical objects.

The Design Task

Let E be an environment and let G be a predicate defined for E. Let a cognitive agent have a goal to have G be true in E. This sets up a design task: to specify an object O, and specify a way to embed O in E, such that when O is so embedded, G is caused to be true.

Traditional definitions of the design task focus on the need to specify the object, e.g., to provide a list of components from some component library and a way of composing them. Our definition additionally requires that a way of embedding O in E be specified; the design task is not complete until the designer specifies a *mode of deployment* of O. The mode of deployment makes the connection between the properties and structure of O and the achievement of G in E. Specifying the mode of deployment becomes necessary if G is defined without any commitment to the properties or structure of O.

The definition of a function should not make any reference to the structure of the object that has the function. Consider the example of a buzzer. In the literature, the function is typically stated as "when the switch is pressed, a sound is made," which makes reference to the switch, a part of the structure of the device. It would be better to give the buzzer function to the designer simply as:

no sound in the environment

$\longrightarrow$ buzzing sound in the environment

It is useful to think of the definition of buzzing as potentially existing independently of, and prior to, the design of the buzzer. By isolating the function definition from any reference to the structure, we are leaving it open for the designer to come up with a very different object to achieve the function. Perhaps one design would achieve the function when it is twisted, another when it is blown on, and so on.

Ontology: an object, in an environment, viewed from a perspective

The world is composed of objects in causal interaction with each other. The primitive representational notion for us is that of *an object, in an environment, viewed from a perspective.* Representationally, the basic elements are:

<object> in <view>
  <object properties>
  <generic environmental properties in
    potential causal relation with object>
  <property relations>

An object in the real world has an open-ended number of properties: science can discover new properties or relationships between existing properties, and one can define new properties from old properties. A *view* is a specific modeling stance; it selects certain properties of the object for representation. The view also implicitly specifies the classes of external objects with which an object can be in causal interaction.

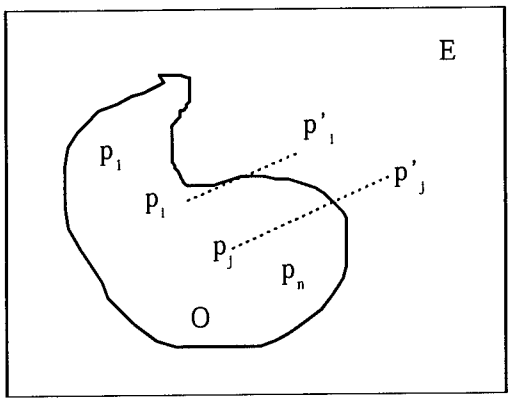


Figure 1. An object O in a generic environment E. Object properties p_i and p_j are in causal interaction with generic environmental properties p'_i and p'_j. When the object is embedded in a specific environment, its mode of deployment is specified by property relations.

The central idea is illustrated in Figure 1. An object interacts with its environment because some of its properties either affect or are affected by the properties of objects in the environment. When two people are in a room, what one person says affects the mood of the other person. When an electrical wire comes in contact with an electrical terminal of an object in its environment, depending upon which of the voltages is the independent variable, the voltage of one of the terminals causes the voltage of the other terminal to have the same value. The terminals are simply special cases where the property is localized to a physical location, but a more general way of talking about causal interaction between objects is by means of the properties that causally interact. When we wish to describe the object's potential interactions in some generality, the environment is specified in general terms. That is, the environmental properties that the object can interact with are described by their types.

Finally, a set of *property relations* is given that represent the modeler's causal understanding of the object. The relations state all the causal relations between the properties, both the object's and environmental ones, believed by the modeler to be relevant. The property relations can be in any form: continuous, discrete, qualitative, etc.

Defining Functions

The central idea we propose for defining functions is that *function of an object is the effect it has on its environment.*

Definition. *Function.* Let G be a formula defined over properties of interest in an environment E. Let us consider the environment plus an object O. If O (by virtue of certain of its properties) causes G to be true in E, we say that D performs, has, or achieves the *function* (or *role*) G.

A description of how O is used to achieve the function (or serve the role, etc.) G has three parts:

1. Functional formula expressing G: what predicate of the environment will be true, under what conditions.

2. Description of properties: what properties of O are used in achieving G.

3. Mode of deployment: what property relations (using what properties of the environment) determine the causal interactions between the object and its environment. This is commonly given by specifying the types of connections between an object and objects in its environment.

Example. Pump: The properties of interest in the environment are the quantities of water, $Q_1(t)$ and $Q_2(t)$, at time t, in locations L_1 and L_2 respectively. Let G be the formula corresponding to $Q_1(t_0) - Q_1(t_f) = Q_2(t_f) - Q_2(t_0) = K > 0$. That is, a positive quantity of water is moved from L_1 to L_2 from the initial instant to final instant. For simplicity let us call this formula Pump(K, L_1, L_2).

Note that while G is described as a function of object O, both preconditions and effects in the specification of G are defined exclusively in terms of properties outside of O. The function of an object is the effect it has on its environment, not its behavior in isolation. In the Pump example, the formula Pump(K, L_1, L_2) describes an effect on the environment. If an object is introduced that causes the formula to be true, we will say that the object "plays the role of a Pump" or "has a Pump function." A particular pump, P, say a reciprocating pump that uses a piston to repeatedly move equal units of water, has relevant properties of having an inlet port $Port_1$ and an outlet port $Port_2$, and is deployed by having $Port_1$ connected to L_1 and $Port_2$ connected to L_2 so that (water at $Port_1$) = (water at L_1) and (water at $Port_2$) = (water at L_2)

This definition of function applies to both intended functions and natural functions. For example, if we have a goal of making the formula Pump(K, L_1, L_2) true, and we design a device which, when embedded in the environment, causes the formula to be true, then we say that the device has the intended function Pump. Applying the definition with appropriate locations L_1 and L_2, we can also say that the heart has a Pump function in the body. The definition of function is neutral with respect to whether the cause-effect description is intended or is described after the fact.

This definition of function applies to both to static and dynamic objects. For static objects, the object that has function F causes F to be true of the environment E when it is appropriately embedded in E. The flower-arrangement, when placed in the room, causes the predicate *Pleasant* to be true of the room. The chair, when it is in a certain relation to the person sitting in it, causes the sitter's bottom to be supported comfortably. For dynamic objects, describing the role or function of an object will typically require giving a sequence of states of the environment. Let us consider an example: The Automated Teller Machine is an example that has been much used in the object-oriented design community. The description of the function *Customer-Withdraw-cash* (k) can be given as:

{Customer-cash = x, balance-in-customer-account = y, y > k } ⟶
{Customer-cash = x + k, balance-in-customer-account = y - k }

The intended interpretation is that the antecedent is true at t_0 and the consequent at t_f.

Note that in this example both the antecedent and the consequent in the functional description are external to the object. No mention is made of any aspect of the structure of the object, such as "If *User_action* = *Push* at location *switch-button*,..." While *User_action* = *Push* is an entirely environmental property, and as such satisfies the requirements to participate in the function description, this is only meaningfully in interaction with a specific location of the object, *switch-button*. Not mentioning any structural feature of the functional object allows the desired function to be expressed prior to choosing or designing an object to fulfill the function.

Composing Objects

It is attractive to imagine design activity that uses a library of stored designs and proceeds by specializing and composing items from the library. When we connect two objects, we are making it possible for selected properties of the two objects to be in causal interaction of the type determined by the type of connection. Thus, representationally, connecting two objects involves declaring which properties of the two objects are in causal interaction. We can identify types of connections and associate with each type the properties whose causal interactions are enabled. For example, being in physical proximity is one type of connection which enables magnetic and thermal properties to interact. Being in physical contact is another type which enable properties associated with force, motion, etc. to interact. Our basic ontology for an object is not one of the object in isolation, but in some environment, in contact with other objects. Composing objects is describing how each becomes part of the other's environment. They can be conceived as causally connected only if they are compatible.

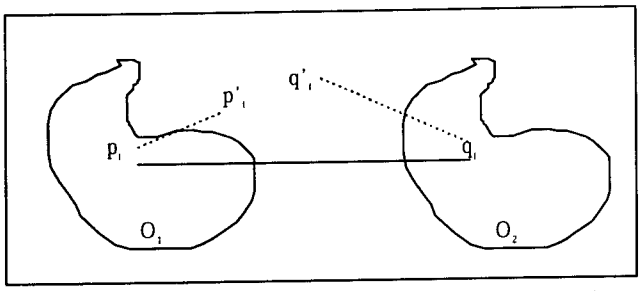


Figure 2. Composition of two objects. If qi is of the type p'_i, and p_i is of the type q'_i, then O_1 and O_2 have a compatible connection for properties p_i and q_i.

Example. Composing two resistors. Consider a resistor with a representation as follows.

Object: *resistor*
Intrinsic Properties:
v_1, v_2, type voltage, at terminals p_1 and p_2
I, type current
R, type resistance
Environmental Properties:
v'_1 , type voltage, in causal interaction with v_1
v'_2, type voltage, in causal interaction with v_2
Property Relations:
$v'_1 = v_1$, $v'_2 = v_2$
$I = (v_1 - v_2)/R$

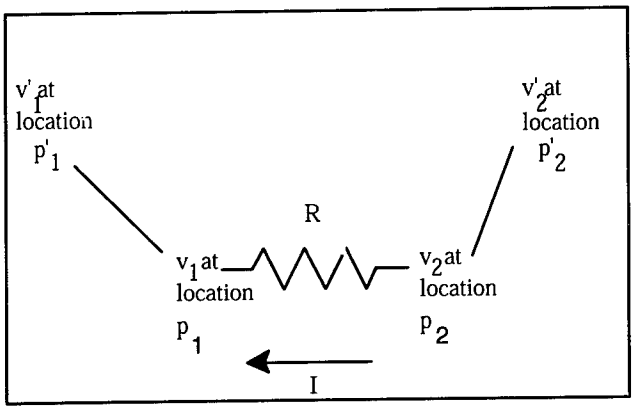


Figure 3. Resistor. v_1 and v_2 are voltages at terminal locations p_1 and p_2 respectively, in causal interaction with voltage properties v'_1 and v'_2 of terminals in the environment. I is current; R is resistance.

There are four ways that two such objects can be connected (within the compatibility requirements), two of them serial and two parallel. Below, we give one instance for each type. (To represent two different resisters we use an additional subscript, replacing variables R, v_1, v_2, p_1, p_2, I, v'_1 and v'_2 by R_i, v_{i1}, v_{i2}, p_{i1}, p_{i2},I_i, v'_{i1} and v'_{i2}, for i = 1, 2.)

Serial. v_{12} in causal interaction with v_{21}. This calls for setting $v'_{12} = v_{21}$, and $v'_{21} = v_{12}$.

Parallel. v_{11} in causal interaction with v_{21} and v_{12} in causal interaction with v_{22}. This requires setting $v'_{11} = v_{21}$, $v'_{21} = v_{11}$, $v'_{12} = v_{22}$, $v'_{22} = v_{21}$.

Composite object in a new view

Our minimalist ontology gives quite a bit of support for generating a description of a composite object from descriptions of individual objects and their connections. The details are somewhat complicated, however, and not directly relevant to the main points of this paper.

After deriving a representation of the composite object, we might wish to re-represent the composite object in a new view, by suppressing some of the component-level properties, introducing new property abstractions, and by restricting our representation of its causal interaction with the external world. In the case of the serial resistors, the modeler might wish to suppress the identity of the individual resistors and make only v_{11} and v_{22} available for external interactions. In this case, the composed object will be represented in a new view, where the object properties are simply R, v_1, v_2, and I, and the external properties are simply the two voltages of objects connected to the two terminals of the composed resistor. Generating this sort of reduced representations for certain kinds of composite objects has been discussed in the literature on Consolidation [2].

For another example, consider an electronic *Adder* circuit. Composing its components, we can generate a description of it in terms of voltages and currents and generate a set of property relations involving both object and external properties. It can also be represented in the *Adder* view: instead of voltages and currents, new property abstractions of *addends* and *sum* and their interrelations would describe the composed object. This would typically be the user view of the *Adder* object.

An example that is especially interesting occurs when a new state variable is created from a behavior trajectory. Let *s* be a numerical state variable. A Boolean state variable *oscillating* can be defined based on whether the behavior trajectory of *s* is of the form {0, 1, 0, -1, 0, ...}, with *oscillating* being true when the trajectory of *s* satisfies the form.

Let us consider the composite object formed from the series composition of an electrical switch, a battery, and a heater-resistor. We give two representations, one which retains all the properties of the components (except that further external electrical connections are not included), and one in a new view that we call the "user view." The user view suppresses the electrical properties.

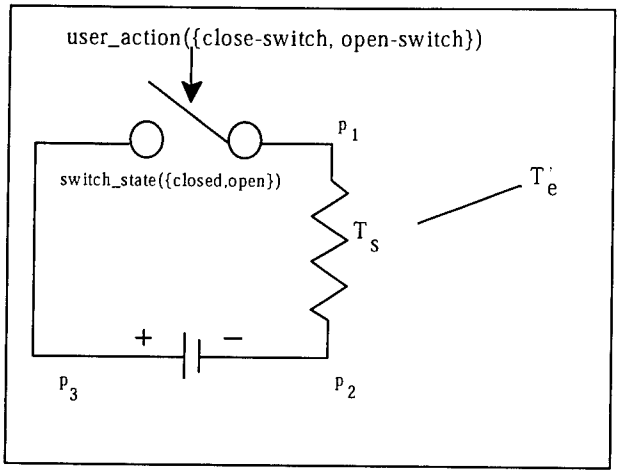


Figure 4. Electrical circuit, with objects *electrical_switch*, *battery* and *heater-resistor* composed in series. We show three electrical terminals p_1, p_2, and p_3, with voltages v_1, v_2, and v_3 as voltage properties. The device has a physical terminal in interaction with the environmental property, user_action, and one thermal terminal, with property T_s, the surface temperature, in interaction with an external fluid layer with property T'_e, its surface temperature.

Composed-Object: *Resistor circuit* (in a view that retains the components' properties).
Intrinsic properties:
voltages v1, v2, and v3 at terminals p1, p2 and p3
I, current through the circuit
R, B, b, k, T_s (resistance, battery voltage, internal resistance of the battery, conversion constant from electrical to thermal energy, and the surface temperature of the heater-resistor, respectively)
Environmental properties:
User_action ({close-switch, open-switch})
T'_e, temperature of fluid layer in contact with resistor surface
T_a, ambient temperature beyond the immediate layer in contact with resistor surface
Property relations:
If User_action = close-switch, then
Switch_state = closed
$I = B / (R+b)$, $v_1 - v_2 = R \cdot I$
$T_s = T_a + k \cdot (I^2)$, $T'_e = T_s$ ($> T_a$)
If User_action = open-switch, then
Switch_state = open
$I = 0$, $T'_e = T_s$ ($= T_a$)

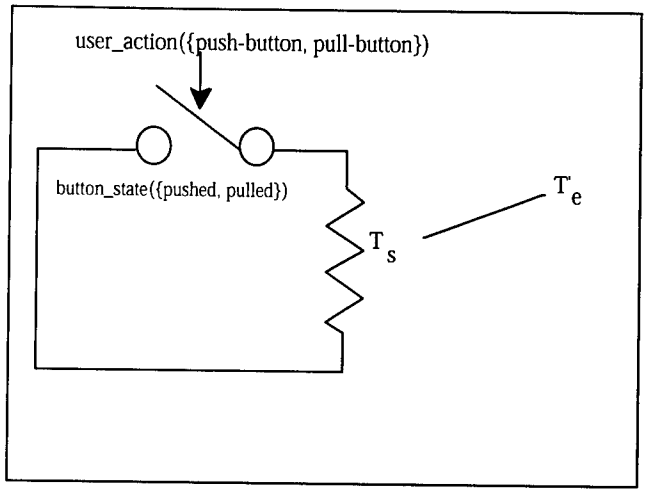


Figure 5. Heater. This is a user view that suppresses electrical properties. There are only three object properties, button_state, heater temperature rating T_r, which is presumed to be greater than the ambient temperature T_a, and heater-surface temperature T_s. Button_state describes the physical state as opposed to switch_state, which describes the electrical state. Causal interactions take place through the environmental properties, User_action and T'_e the temperature of the fluid layer in contact with the heater surface.

Object: *Heater* (user view)
Intrinsic properties:
button-state
T_r, heater temperature rating
T_s, heater surface temperature
Environmental properties:
User_action ({push-button, pull-button })
T'_e
Property relations:
$T'_e = T_s$
If User_action = push-button, then
button-state = pushed
$T_s = T_r$ ($T_r > T_a$ the ambient temperature)
If *User_action = pull-button, then*
button-state = pulled
$T_s = T_a$

Functions of the heater and its components

For each of the objects, selected functions are given below. Each function is named and has a functional formula specification, a description of properties, and a mode of deployment.

Electrical switch
Functions: *close_connection* (p'_1, p'_2)
open_connection (p'_1, p'_2)
Functional Formulae:

close_connection (p'1, p'2):
v'1 = v'2 (where v'i is the voltage at p'i)
open_connection (p'1, p'2):
I (from p'1 to p'2) = 0
Properties:
Intrinsic: v1, v2 voltages at terminals p1 and p2;
switch_state{Closed, Open })
Environmental:
v'1 ,v'2 voltages at terminals p'1 and p'2;
user_action{close-switch, open-switch }
Mode of Deployment:
p1 electrically connected to p'1;
p2 electrically connected to p'2;
switch_state in causal interaction with user_action so that switch_state = Closed iff user_action = close-switch

Battery
Function: *Apply_voltage(B) across p'1 and p'2*
Functional Formulae: v'1 - v'2 = B
Properties:
Intrinsic: v1 and v2 voltages at terminals p1 and p2
Environmental: v'1 and v'2, at terminals p'1 and p'2
Mode of Deployment:
p1 electrically connected to p'1;
p2 electrically connected to p'2

Heater-resistor
Function: *Heat_Fluid_Surface* (T'e > T'e0,)
Functional Formulae:
When v'1 - v'2 = 0,
T'e = T'e0 (initial fluid-surface temperature)
When v'1 - v'2 = va > 0,
$T'_e = k \cdot (v_a/R)^2 + T'_{e0}$
Properties:
Intrinsic: v1 and v2 voltages at terminals p1 and p2;
Ts temperature of surface;
k constant for resistor's transformation of electrical to heat energy
Environmental:
v'1 and v'2, voltages at terminals p'1 and p'2;
T'e, fluid-surface temperature
Mode of Deployment:
p1 electrically connected to p'1;
p2 electrically connected to p'2;
Ts = T'e (fluid surface in thermal contact with resistor surface)

Heater (user view)
Functions:
Heat_Fluid_Surface (Tr)
Not_Heat_Fluid_Surface
Functional Formulae:
Heat_Fluid_Surface (Tr): T'e = Tr > T'e0
Not_Heat_Fluid_Surface: T'e = T'e0
Properties:
Intrinsic: button-state {pushed, pulled};
temperature rating Tr; surface temperature Ts
Environmental: fluid surface temperature T'e,
ambient temperature Ta;
user_action {Push-button, Pull -button}
Mode of deployment:
Ts = T'e;
button_state in causal interaction with user_action so that button_state = pushed iff user_action = Push-button

Explaining How a Function is Achieved

An important requirement for a functional framework is that it should support reasoning about the relationships between the properties of an object and those of its components. This is essential for both diagnostic and design problem solving.

The stream of work on functions called Functional Representation (FR) Language (summarized in [3]) focuses on the relationships between the functions of a device and its structure. In particular, it proposes a representation called a causal process description (CPD) to explain how the device achieves the function. In the CPD, the causal transitions are associated with formally interpretable explanatory annotations. The annotation that links the device level to the component level is one that explains a transition by appealing to some function of a component. This way of explaining has intuitive appeal.

However, in the FR work, function is defined as a sort of abstraction of an object's own behavior. Thus, in its definition of function, it does not make the distinction between the object's behavior and the object's effects on its environment that we make here. It will be useful to show that the intuition of explaining a device level function in terms of component functions can be supported when we adopt the definition of function proposed here. However, we do not show it in this short paper.

Related Work and Discussion

There has been an explosion of work on functions in recent years, so we will only discuss, and that briefly, work directly relevant to the issues central to the current paper. A substantial body of work can be thought of as content-theory that can fit comfortably with the notions developed here. A set of basic roles in mechanical interactions is provided in [6] and, in a somewhat different subdomain, in Goel [19], roles that components of loops play in algorithms are given in [7], roles that seem to be common

to different domains sharing the ontology of flows is given in [8] et al... The focus on functional roles common to flow systems is shared by [9] and [10]. There has been a substantial body of work in visual understanding (see, e.g., [11]) wherein recognitional algorithms use a functional rather than a structural definition of objects to be recognized. Thus a recognizer for chair would see if the object can in fact support the sitting of a person rather than look for specific subparts.

In the pioneering work on function in diagnosis [12], the function was closely tied to the object. The work in the FR tradition (summarized in [3]) and on CFRL [1, 13] [14] treats function as an abstraction of the behavior of a device, as does [15]. This is entirely adequate for many purposes, but the separation of function from the object's properties will help those investigations to reach wider applicability. Recently, [16] and [17] share some essential intuitions with the present paper in that they make an effort to separate the behavior of an object from its function as the role played in the achievement of a designer's goal. The present paper proposes what appears to be the simplest ontology in which this notion can be explicated, and also makes various kinds of unifications, such as between static and dynamic objects and between intended functions and functions observed in natural systems. Research on function types of [18] is orthogonal to the work here, and can be restated to be consistent with the definition of function proposed here.

Locating the function in the effects on the environment, rather than in the object, clarifies the notion of multiple realizability of functions. When the function is defined without any reference to the structure of an object, different realizations of the function become possible.

This multiple realizability also suggests criteria by which one could decide at what level of organization the effects of an object should be described. For example, we may describe the role of the thermostat as "When the room temperature is below T_{set}, the furnace is on." Note that both predicates are of objects in the environment. However, an effect of the thermostat so configured is also eventually to make a person in the room warm. Does this mean that it is equally plausible to attribute to the thermostat the function, "make a person warm"? Suppose that "make a person warm" can be multiply realized, for example, one, by covering the person with a woolen blanket and two, by using a thermostat-controlled furnace. The thermostat's effect is actually on the furnace and the thermostat-furnace configuration as a whole has the effect of keeping the person warm. Thus, in the given modeling context, the thermostat's function is best stated as its role in linking the temperature of the room to the starting of the furnace.

Need to generalize causality beyond property relations

It is attractive to represent properties as simply variables associated with objects that enable them to interact causally with other objects and properties of their environments. Then causality can be represent as property relations, which establish dependencies between variables. This sets up an attractive formalization for causal relationships as mathematical functions expressing relations of properties.

Yet it is difficult to see how to represent all causal relations as property relations and all property relations as mathematical functions. For one thing, objects can be created and destroyed; examples include antibodies, casting molds, and action plans. Further, a configuration might change, e.g., an object gets out of alignment. In any case, the object ontology we have described seems not to be sufficient to handle descriptions like, "heat applied to water causes boiling." "Water" is not quite an object in the sense of the simple object-property-environment-view ontology, and "boiling" is a process, which is unclear how to represent. The notion of "property relations" needs to be generalized to include causal relations of all sorts. Perhaps a better term would be "causal relations" whereby any proposition standing for a state of affairs (SOA) can be causally dependent on another SOA, where a SOA can include properties, configurations, the existence of objects, the occurrence and properties of processes, and whatever other entities there are that participate in relationships of causal dependence.

Acknowledgments

We thank Susan Josephson for several good discussions on the topic of this paper. The research for this paper was supported by DARPA under two contracts: one contract no. F30602-93-C-0243 monitored by USAF Materiel Command, Rome Laboratories, DARPA order no. A714, and by grant no. N00014-96-1-0701, DARPA order no D594.

References

[1] Y. Iwasaki and B. Chandrasekaran, "Design Verification Through Function- and Behavior-Oriented Representation: Bridging the gap between function and behavior," in *Artificial Intelligence in Design '92*, J. S. Gero, Ed.: Kluwer Academic Publishers, 1992, pp. 597-616.

[2] T. Bylander, "A Critique of Qualitative Simulation From a Consolidation Viewpoint," *IEEE Trans. Systems, Man and Cybernetics*, vol. 18, pp. 252-263, 1988.

[3] B. Chandrasekaran, "Functional representation and causal processes," in *Advances in Computers*, vol. 38, M. C. Yovits, Ed.: Academic Press, 1994, pp. 73-143.

[4] Y. Iwasaki, and H. A. Simon, "Causality in device behavior," *Artificial Intelligence*, vol. 29, 1986.

[5] H. A. Simon, "Causal ordering and identifiability," in *Studies in Econometric Methods*, T. Koopman and W. Hood, Eds. New York: John Wiley and Sons, 1953.

[6] J. Hodges, "Naive mechanics: A computational model of device use and function in design improvisation," *IEEE Expert*, vol. 7, pp. 14-27, 1992.

[7] D. Allemang, "Using Functional Models in Automatic Debugging," in *IEEE Expert*, vol. 6, 1991, pp. 13-18.

[8] L. Chittaro, C. Tasso, and E. Toppano, "Putting functional knowledge on firmer ground," *International Journal of Applied Artificial Intelligence*, vol. 8, pp. 239-258, 1994.

[9] M. Lind, "Modeling goals and functions of complex industrial plants," *Applied Artificial Intelligence*, vol. 8, pp. 259-283, 1994.

[10] A. N. Kumar, and S. J. Upadhyaya, "Function-based discrimination using model-based diagnosis," *International Journal of Applied Artificial Intelligence*, vol. 9, pp. 65-80, 1995.

[11] L. Stark, and K. W. Boyer, "Achieving generalized object recognition through reasoning about association of function to structure," *IEEE Trans. Pattern Analysis and Machine Intelligence*, vol. 13, pp. 1097-1104, 1991.

[12] R. Davis, "Diagnostic Reasoning Based on Structure and Function," *Artificial Intelligence*, vol. 24, pp. 7-84, 1984.

[13] Y. Iwasaki, R. Fikes, M. Vescovi, and B. Chandrasekaran, "How Things Are Intended to Work: Capturing functional knowledge in device design," in *Proceedings of the 13th International Joint Conference on Artificial Intelligence*. Mountain View, CA: Morgan Kaufmann, 1993, pp. 1516-1522.

[14] M. Vescovi, Y. Iwasaki, R. Fikes, and B. Chandrasekaran, "CFRL: A Language for Specifying the Causal Functionality of Engineered Devices," in *Eleventh National Conference on AI*: AAAI Press/MIT Press, 1993, pp. 626-633.

[15] Y. Umeda, H. Takeda, T. Tomiyama, and H. Yoshikawa, "Function, behavior and structure," in *AI in Engineering*: Computational Mechanics Publications and Springer Verlag, 1990, pp. 177-193.

[16] M. Sasajima, Y. Kitamura, M. Ikeda, and R. Mizoguchi, "FBRL: A function and behavior representation language," in *International Joint Conf on Artificial Intelligence*, vol. 2. Montreal: IJCAI, Inc. and Morgan Kaufmann, 1995, pp. 1830-1836.

[17] J. E. Larsson, "Diagnosis based on explicit means-end models," *Artificial Intelligence*, 1995.

[18] A. Keuneke, "Device Representation: The significance of functional knowledge," in *IEEE Expert*, vol. 6, 1991, pp. 22-25.

[19] A. K. Goel, "Integration of cased-based reasoning and model-based reasoning for adaptive design problem solving, Ph.D. Thesis, Ohio State University, 1989.

Proceedings 5th Annual Dual-Use Technologies & Applications Conference, IEEE and Rome Lab, Utica, NY, May 22-25, 1995, 462-467.

Functional Representation and Understanding of Software: Technology and Application

John Hartman B. Chandrasekaran

Laboratory for Artificial Intelligence Research, Dept. of Computer and Information Science
The Ohio State University, 2015 Neil Ave., Columbus, OH 43210-1277
http://www.cis.ohio-state.edu/hypertext/LAIR/lair-page.html
hartman *or* chandra@cis.ohio-state.edu

Abstract

Government and industry spend over $100 billion a year to preserve and extend existing software. Existing tools have shallow analysis and limited impact. Improved tools with deeper, human-like program understanding will reduce the huge cost of activities involving existing software.

We describe our automatic program understanding theory and technology. Our approach has two parts: understanding and representation. The UNPROG system uses programming plan knowledge to recognize deep programming concepts in existing programs. Functional Representation (FR) is a theory and language for representing understanding of devices, including programs. We use FR to capture program understanding to give the explanations required by applications.

Automatic reverse engineering and reengineering tools can use this understanding to produce more useful program descriptions and reengineered code. We describe what has been accomplished so far, and discuss how this strategic dual-use technology can be further developed and applied.

1 Introduction

Perhaps $5 trillion is invested in existing "legacy" software. Government and industry spend over $100 billion annually to preserve and extend existing software.[1] Although much of this work is conducted in-house, the current maintenance service and tools market is estimated to be $15 billion, and rapid growth is expected.

Much of maintenance (and programming) involves understanding programs. Current tools for existing programs have limited effectiveness and impact because their analysis is shallow. Improved tools with deeper, human-like program understanding will have greater acceptance and value, reducing the huge cost of activities involving existing software.

This paper describes how automatic program understanding (APU) will improve software tools. Our theory and technology has two parts: understanding and representation. Automatic program understanders recognize abstract concepts like "read-process" and "hash table" in existing programs. Functional Representation (FR) represents the program's function in terms of component functions found by APU. This gives explanations and explanation structure which can be exploited by many applications.

In this paper we will briefly: 1) describe automatic program understanding and the UNPROG program understander, 2) introduce Functional Representation, and show how it produces explanations using concepts recognized by UNPROG, and 3) demonstrate and discuss how this enabling tool technology can be applied, developed and commercialized.

2 Automatic Program Understanding

Automatic program understanders use programming knowledge to recognize abstract concepts in programs. First we briefly review plan-based program understanding. Then we describe the UNPROG program understander.

[0]This work was supported by ARPA, Order No. A714, monitored by USAF Materiel Command Rome Laboratories–Contract F30602-93-C-0243.

[1]Most of the $20 billion annual Federal expenditure is by the Department of Defense.

2.1 Plan-Based Understanding

Plans are units of programming knowledge connecting concepts and their implementations.[6][2] Recognizing plans used by the programmer can recover his abstract concepts and intentions.

Suppose a programmer needs to read and process employee data in his payroll program. He doesn't have to reinvent "read-process" because this concept is part of his programming knowledge, along with plans used to implement the concept under different constraints. The PAYDAY program is the result of implementing "read-process" with a "read-process loop" plan consisting of a particular loop form terminated by a signal to a control variable:

```
N := ZERO;
loop
     GET(SSN); GET(JOB); GET(PAY);
     PUT(SSN); PUT(JOB); PUT(PAY);
     N := N + 1;
     exit when SSN < ZERO;
     if job < 5 then
          if job <  2 then
               goto PRINT_PAY
          else
               goto DED2;
          end if;
     end if;
     DEDUCT := PAY * .2;
     goto PRINT_DED;
<<DED2>>      DEDUCT := PAY * P15;
<<PRINT_DED>> PUT(DEDUCT);
<<PRINT-PAY>> PUT(PAY);
end loop;
N := N - 1;
PUT(N);
```

Figure 1: PAYDAY Source Program

In Figure 2, an automatic or human understander is trying to understand PAYDAY. The understander

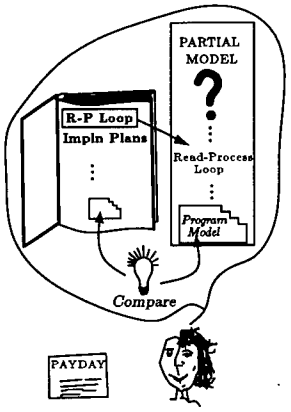


Figure 2: Understanding With A Plan

also has programming knowledge about reading and processing data, including some of same plans as the programmer. She forms a program model and searches it for instances of implementation plans. Here she finds one and says, "Aha! A 'read-process loop with control variable' plan!! This must be 'read-process'!!!" She recovered an abstract concept that was in the mind of the programmer, but not explicit in his source code.

Automatic program understanders recognize abstract concepts like "read-process", "hash table" and "sorting" in source programs. APU's exploit the knowledge and reasoning processes human programmers use to understand programs, especially plans. Most existing APU's are impractical research systems built to study difficult recognition tasks.[3]

2.2 UNPROG Understander

In contrast, the UNPROG automatic program understander is designed to investigate program understanding and its applications with real-world programs.[4] UNPROG uses plan knowledge to efficiently recognize control concepts. Control concepts are abstract notions about the interaction of control flow, data flow and computation, eg. "read-process", "bounded-linear search" and "do loop".

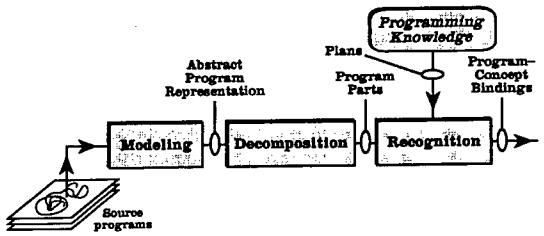


Figure 3: UNPROG Program Understander

As shown in Figure 3, UNPROG recognizes concepts by comparing program parts with standard programming plans from a library of programming knowledge. Source programs are analyzed to form an abstract language-independent program representation. The representation is decomposed into a tree of small program parts by proper decomposition.

Figure 4 shows how UNPROG compares PAYDAY's loop part with a plan for implementing "read-process" using a middle exit loop and input termination signal. The program part is represented with abstract control and data flow (top). The plan is represented by control and data flow schemas, and additional qualifications (bottom). Here the program part and plan can be uniquely bound. Therefore, UNPROG recognizes "read-process", its implementation, and associated concepts. Its output is bind-

[2]The authors' papers are available from the www address in the title.

[3]APU surveys are in [5] and [7].

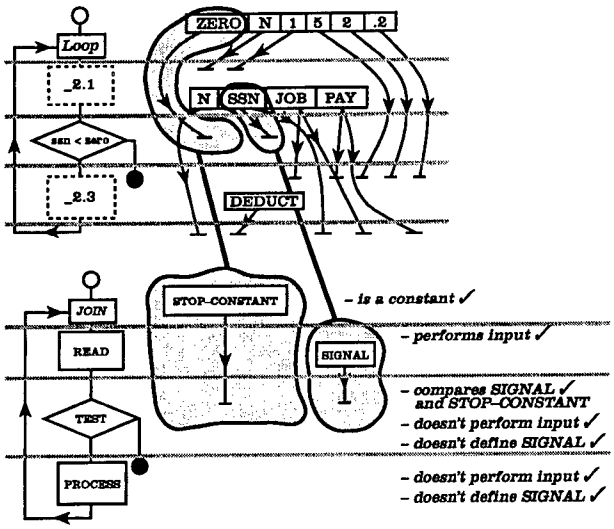


Figure 4: Recognizing "Read-Process"

ings and correspondences between the program, plan and concepts.

UNPROG is a powerful technology for identifying deep programming concepts, regardless of how they are implemented. However, UNPROG is neutral about how such understanding is represented and used.

3 Functional Representation of Programs

Functional Representation is a general theory about representing understanding. In this section we will briefly review Functional Representation and functional representation of program understanding. We will show how we are applying it to capture and use the understanding created by UNPROG to support particular explanations and applications.

3.1 Functional Representation

Functional Representation is a theory and language for reasoning about functionality and causal processes in devices. It has been successfully applied to a large variety of tasks and devices, eg. explaining failures in a chemical plant, medical diagnosis, and engineering design verification.[2]

The main ideas of FR, as they relate to software, are:

Formal Specification The functions of devices and their parts are given by formal specifications describing possible states.

Causal Process Descriptions Processes in devices are described as transitions between specified states.

Abstraction Levels Function and transitions ("what") are described in terms of behavior and other explanations at lower levels ("how").

Components and Structure Function is shown to emerge from structure consisting of a particular composition of components.

3.2 Representing Understanding

Allemang showed how Functional Representation can capture particular reasonings or explanations about a program.[1] In his work, an FR is a proof or argument structure that applies to a large class of programs. Therefore it can be reused for programs that are found to be members of the class. The same is true for parts of FR's which apply to parts of programs. Therefore FR can be used to represent plans and concepts, their semantics, and their consequences. Allemang formalized this use of Functional Representation as program *functional semantics*. He demonstrated its advantages over traditional programming language semantics for certain kinds of reasoning about programs.

We are now addressing practical understanding and applications using UNPROG and FR. UNPROG gives the technology to efficiently recognize plan and concept instances in real-world programs. FR gives the representation for this understanding and its use, formally grounded by functional semantics.

In the example above UNPROG recognized "read-process" and associated concepts in PAYDAY. With UNPROG output represented in FR, many explanations and applications are possible. For example, given data dictionary information, the documentation in Figure 5 can be generated automatically.

```
A read-process loop terminated by a negative signal value
in SSN reads and processes employee data. For each employee:

    The READ block:
      (1) reads and echoes SSN, JOB and PAY,
      (2) counts the number of employees processed (N).

    The PROCESS block:
      (1) determines and prints the deductions
            (DEDUCT), if any, using a logic network,
      (2) prints PAY.

The number of employees processed (N) is printed.
```

Figure 5: Automatic PAYDAY Documentation

Automatic question answering is another application, which requires a different kind of explanation. We'll use it to illustrate the functional representation

of PAYDAY. The complete FR is large and contains various abstractions, eg. code to plan, plan to concept, concept to explanation. These abstractions involve various abstraction principles, eg. recognition, selection, proof, and involve various languages, eg. programming language semantics, predicate calculus, and English. We've simplified here.

```
Q> What does PAYDAY do?

A: Its functions include HISTORY, OUTPUT,
        and PAYROLL-CALCULATION
```

FR represents PAYDAY as a device with various functions, eg. giving different views or decompositions.

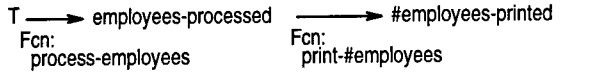
```
Q> What does PAYDAY/HISTORY do?

A: Makes History(employees-processed,
            #employees-printed)
```

Functions have **ToMake:** slots giving their postconditions in some state language. The state description in the answer means that the function produces a temporal history where state `employees-processed` was reached and then state `#employees-printed` was reached.

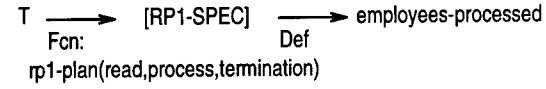
```
Q> How did PAYDAY/HISTORY make
 History(employees-processed,#employees-printed)?
```

FR expresses functions' behaviors with causal process descriptions (CPD's). CPD's are state transition diagrams connecting preconditions and postconditions. Their links are annotated with justifications for the transitions, eg. functions, sub-CPD's, or non-causal links such as definitions. The CPD giving PAYDAY/HISTORY's behavior is:



This shows that `employees-processed` is caused by the PROCESS-EMPLOYEE function.

PROCESS-EMPLOYEE's behavior is given by its CPD:



The first transition is justified by a function of a particular read-process plan, eg. the read-process loop plan used by UNPROG above. The state reached, `[RP1-SPEC]` is a formal specification of a read-process concept.[4] It is provable with the plan instantiated with PAYDAY program parts in its READ, PROCESS and TERMINATION slots. The second transition is a non-causal definition link connecting the string "employees-processed" in the informal discourse language with the formal specification.

In summary, Functional Representation provides a formal representation of program understanding based on causal description, useful abstractions, and component structure. Representing UNPROG output in FR allows many applications to exploit understanding using these principles.

4 Application

Our approach to developing and applying UNPROG, FR, and APU assumes that organizations have particular needs involving their existing software. For example, imagine a DoD organization mandated to translate old systems to Ada, or an insurance company converting to C++. The organization will investigate how this costly task can be automated.[5]

The contribution of existing reverse engineering and reengineering tools is limited because their analysis is shallow. For example, language translators may produce syntactically correct code, but it will have poor human and performance quality because underlying concepts are not recognized and preserved.

Organizations and tool developers should ask,

> What concepts would improve the task if they could be automatically recognized?

Automatic program understanding can provide benefits for many tasks, tools, and concepts. It is an economically important dual-use technology for internal use and for commercial products that work on existing software. In this section we briefly describe the application and commercialization of automatic program understanding.

4.1 Reverse Engineering and Reengineering Tools

Reverse engineering consists of understanding software to form program descriptions needed for particular tasks. Important classes of reverse engineering tools are analyzers, browsers, and inspectors. Reengineering consists of creating new programs to meet new needs. It combines reverse engineering and reimplementation. Important classes of automatic reengineering tools are reformatters, restructurers, converters, and translators. Existing tools use only

[4] The specification says that a history is produced in the form: $R(I_1), P(R(I_1)) \cdots R(I_n)$ with $\neg T(I_i), 1 \leq i \leq n-1$ and $T(I_n)$. R and T are functions performed on input items I_i, and P is a function performed on the results of R.

[5] A typical out-source conversion price is $10/line.

syntactic information and weak general methods. They produce shallow descriptions and poor code with degraded human factors and performance. Recognizing deeper concepts will increase tool performance and value.

Automatic program understanding can be developed and applied, and benefits can be quantified, using the *tool improvement paradigm* shown in Figure 6.

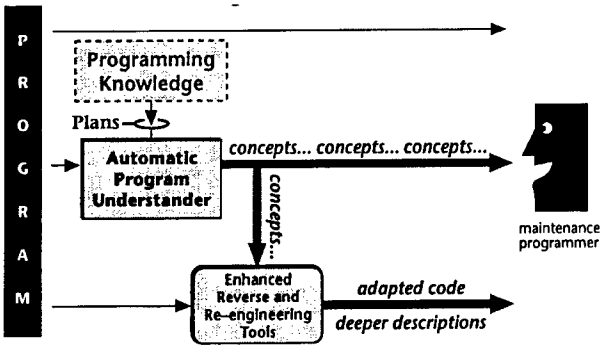


Figure 6: Tool Improvement

In this paradigm an organization identifies a tool's limitations for a task. Concepts are found which overcome those limitations when recognized. An automatic program understander and needed knowledge are developed to recognize instances of these concepts in the program population. Recognized concepts by themselves help the programmer perform the task. More importantly, the tool is modified to produce improved output using recognized concepts.

For example, translation to Ada can be tailored to concepts such as "read-process". The resulting concept-specific translation is clearly an improvement over syntactic translation. Recognized concepts will be preserved and highlighted instead of being destroyed and obscured. Code style and performance can be more closely tailored to the language and task. Additional possible benefits include documentation, reuse, formal specification, and entry into CASE.

We demonstrated this paradigm for an important reengineering task and tool. Restructuring translates programs with unstructured control flow graphs to structured graphs, eg. for improved maintenance or translation to structured languages. Commercial restructurers produce code which, though technically structured, is larger, stilted, obscure, and less efficient.

RESTRUC uses concepts recognized by UNPROG to produce restructured code that has quality which cannot be produced with existing syntactic methods. RESTRUC uses concepts recognized by UNPROG to: 1) perform strong, concept-specific structuring transformations, 2) generate insightful code at the program, statement, and format levels, and 3) add documentation, annotation, and other benefits. Here is part of RESTRUC's PAYDAY output:

```
--      --- Read-Process Loop ---
-- Terminated by signal variable: SSN,
--                  stop-constant: ZERO
-- Transformed from middle-exit loop by:
--   signal loop-once initialization (RP1-8)

     SSN :=  LOOP_ONCE;
while not (SSN < ZERO) loop
--        --- Read ---
        GET(SSN); GET(JOB); GET(PAY);
        PUT(SSN); PUT(JOB); PUT(PAY);
        N := N + 1;
        if not (SSN < ZERO) then
--       --- Process ---
                P2_3_1;
                PUT(PAY);
        end if;
end loop;
```

Figure 7: Improved Restructuring

The original middle-exit loop must be transformed to a WHILE loop. Existing restructurers do this with general algorithms which necessarily introduce degradations such as new variables and tests, increased complexity, reordered code, and replicated code. In contrast, UNPROG recognized the termination condition in SSN. This was used to produce a WHILE loop preserving and displaying the original role of SSN and its test. This cannot be done without recognizing the concepts of the read-process loop.

4.2 Commercialization

The tool improvement paradigm is a model for APU development and commercialization, as well as for application. Academic and industrial researchers will develop the technology ("push") in cooperation with organizations who wish to reduce the cost of tasks involving existing code ("pull"), and tool developers/vendors. (This model generalizes for many dual-use technology transfer and commercialization domains.)

Technical issues for APU development and commercialization include: 1) understander development, 2) understanding representation, 3) knowledge acquisition, and 4) empirical characterization. These issues can be addressed together in prototype enhanced tool projects.

Understander development problems such as efficiency, representation, decomposition, reasoning, and hierarchical recognition are addressed in the academic literature, but in unrealistic contexts. Practical projects will develop these areas for particular concepts and program populations. Practical

projects will also provide data and constraints for basic research.

Understanding representation communicates understanding to applications. Representations must be designed which are effective (and sufficiently formal) for particular concepts and applications, but also general for other applications. Functional Representation provides a suitable framework. As discussed above, FR provides an explanation structure which connects code to concepts needed for particular applications. A given application's knowledge and control echoes parts of the structure. Since FR provides multiple functional explanations or views, other applications can use shared and distinct parts of the representation.

Plan libraries containing hundreds or thousands of plans are needed to produce the concept recognition rates needed for applications. Acquiring such knowledge is currently difficult and expensive. Tools are needed to create, visualize, edit, debug, organize, and test plans. We have proposed *understander-assisted knowledge acquisition* for concurrently performing these tasks during application development, using an APU and other tools.

Finally, empirical studies are needed to characterize recognition performance and benefit for particular program populations, concepts, applications and tasks. We have developed an APU performance model and measures. An important measure is *plan-fulness*, which describes the recognition rate-cost tradeoff for a program population with various sized plan libraries. Empirical studies using this and other measures are needed to evaluate and predict understander performance. Similarly, empirical studies are needed to quantify the ultimate value of APU applications. In the tool improvement paradigm, the value added to a software task and process can be measured relative to a baseline tool and process.

5 Current Research

As discussed above, we are currently developing Functional Representation of programs and plans. The result will be a system, FR-UNPROG, showing how a specific application benefits from concepts recognized by UNPROG and represented in FR.

We are also applying FR to other software engineering problems. FR can be used to represent and explain software architectures as well as programs. We are investigating architecture with David Luckham's Rapide executable architecture definition language from Stanford.[3] FR-Rapide is a tool to aid architecture prototyping with Rapide. It represents an understanding of an architecture using FR, and generates useful explanations.

We're beginning to investigate design capture and verification. This will involve the executable architecture domain, where designs can be captured in FR and verified against prototype executions and other constraints. It also allows us to extend FR and UNPROG for requirements and domain concepts. Finally, we are working on reuse with Bruce Weide and the Reusable Software Research Group at Ohio State. Their RESOLVE is a framework for formal reusable objects. UNPROG and FR have reuse applications, and RESOLVE, FR and UNPROG share many representation issues, eg. for plans and concepts.

Acknowledgments

This work benefits from Frank Gutowski, The Analytix Group. He drew most of the figures for Analytix proposals. This work also benefits from Bruce Weide, the OSU AI and Reusable Software Research Groups, and those acknowledged in previous publications which provided included material.

References

[1] Dean Allemang and B. Chandrasekaran. Functional representation and program debugging. In *6TH ANNUAL KNOWLEDGE-BASED SOFTWARE ENGINEERING CONFERENCE*, 1991.

[2] B. Chandrasekaran. Functional representation and causal processes. In M. Yovits, *ADVANCES in COMPUTERS*. Academic Press, 1994.

[3] David C. Luckham et al. Specification and analysis of system architecture using Rapide. Forthcoming: IEEE Trans. on Software Engineering.

[4] John Hartman. Understanding natural programs using proper decomposition. *13th INTL. CONF. SOFTWARE ENGINRG.* , 1991.

[5] John Hartman. Technical introduction. *AI and AUTOMATED PROGRAM UNDERSTANDING WORKSHOP*, Tenth National Conference On Artificial Intelligence, 1992.

[6] John Hartman. Plans in software engineering - An overview. OSU Lab. For AI Research, 1994.

[7] Linda Mary Wills. Automated program recognition by graph parsing. Technical Report AI-TR-1358, MIT AI Lab., 1992. Ph.D. Thesis.

Representing Functional Requirements and User-System Interactions

B. Chandrasekaran
Laboratory for AI Research
The Ohio State University
591 Dreese Laboratories
Columbus, OH 43210
Email: chandra@cis.ohio-state.edu

Hermann Kaindl
Siemens AG Osterreich, PSE
Geusaugasse 17, A - 1030 Vienna Austria
Email: kaih@siemens.co.at

Abstract

Specifying the requirements for a new system to be built is a sufficiently important issue in systems engineering that it has become a research area of its own called *Requirements Engineering*. Related to this issue, designing and specifying the interactions of potential users with a system is an important problem in *Human-Computer Interaction*. In this paper, we apply *Functional Representation* (FR) to model functional requirements and user-system interaction, in the process clarifying their mutual relationship.

1. Introduction

It is widely accepted that a clear set of requirements facilitates system design — whether it is a software, hardware or a hybrid system. Requirements specification includes precise description of needed functionalities and required interactions between the user and the system, as well as so-called non-functional requirements (constraining the development process and the developed system). The more precisely and unambiguously these requirements are specified, the better off is everyone involved in the whole process: the customer, the system designers and implementers, and users. The required precision and lack of ambiguity can only be achieved if we have a clear understanding of the kind of things that need to be stated as part of the requirements — a clear identification of what has been called the *ontology* of the situation — and support the ontology by means of a formal representation vocabulary.

In this paper we focus on functional requirements and specification of needed interactions between the user and the system. Such interactions have been a subject of discussion in the literature on software engineering and the design of interactive systems (see for example, [1-4]). We will adapt a representation from a body of work known as Functional Representation (FR) (for a review of this work, see [5]) for specifying such interactions. We build on the work on requirements specification and task modeling using functional ideas reported in [6]. In a recent paper on applying functional representation to software reuse and design [7], requirements of some specified functional prototype from other functional prototypes are specified in its implementation. In contrast, we describe requirements of some user for a complete system to be built.

The outline of our argument in this paper is as follows. We discuss some desiderata for a representational framework for requirements. We discuss a definition of function and its relation to the purposes of a user. Together, these give us some terms for representing functions. The definition that we provide introduces the need for specifying how an artifact is to be used — the way a user is to interact with the device — as an intrinsic part of the task of design. We term this part of design *interaction design*. As interaction design proceeds, the interactions needed can be articulated to varying degrees of concreteness. A particularly common representation of such interactions is through *scenarios*, which capture the series of interactions between the user and the system needed for the function to be achieved. We show that such scenarios can be represented in a manner similar to

the so-called *causal process representations* in the FR literature. We motivate our discussion by using as a concrete example the Automated Teller Machine (ATM).

2. Desiderata for a Framework for Functional Requirements

We use the shorthand FIRQ to stand for functional and interaction requirements. We believe that an FIRQ framework should deal with the following issues to some degree.

1. *Specifying functions.* FIRQ should of course allow the specification of desired functions. In case where it is appropriate, it should also allow situations to be avoided, prevented, etc.

2. *Specifying interactions.* In the design of interactive systems, the customer would like to specify, at the design stage, that the intended functions are to come about as a result of certain interactions between the device or system and its user. FIRQ should support the specification of such interactions.

3. *Should not demand information not likely to be available at design time, but should allow representation of information that is available.* FIRQ are typically given before the design of internal structures and their connections (though a certain amount of it might evolve as a result of interaction between design and requirements modification). This means that FIRQ should not demand knowledge of the system — say its structure — that is not available at the time of requirements specification.

4. *Should allow elaboration and refinement of requirements as interaction design proceeds.* By the same token, FIRQ should allow specification of changing requirements as design proceeds and commitments are being made. As interaction design is performed, a more detailed set of requirements emerges. Thus a framework for FIRQ should ideally support requirements evolution during interaction design.

3. What is a Function?

In much of the work on representing functions, including in the work on FR, function is treated as a property of the object or device, often as some abstraction of a selected behavior of the device. As an example, the function `buzz` of the device `buzzer` might be defined as follows: "When the switch is pressed by a user, a sound is produced in its clapper." Note that, in this description, the switch and the clapper are parts of the buzzer, and the behavior of interest is described in terms of the states of these components or ports of the device. This definition certainly captures certain things we want from the definition of a function: that it expresses an intention of a designer or a user, that it is an abstraction of behavior and so on. However, the function cannot be defined if we do not have the device in the first place. Imagine that the buzzer has not been designed yet and a customer is looking for a device to do what the buzzer helps to accomplish. The customer — we will imagine her to be the user of the device — has a purpose in mind which she would like the device to accomplish or help her accomplish. How is this purpose to be represented?

Clearly, it cannot use aspects of the device not yet designed. One of us has argued, in a recent proposal on the definition of function [8], that a *function or role of an object is an effect it has on its environment.* The function defined in this way is a dual to the purpose of a user. The user intends — has the purpose to cause — a certain effect in her world, and if an object or a device can create the effect, then she may attribute the effect as a function of the object.

Let us assume an environment consisting of some objects. The objects may be specified abstractly and incompletely, as long the state variables of interest to us in modeling are available. (We only consider *dynamic functions* here, i.e., functions defined in terms of state variables. There are also what one might call *static functions*, such as the seating function of a chair or the light-passing function of a window, that are defined in terms of objects' static properties. Such functions are discussed in [8], but we do not consider them further here.)

Functions and Purposes. A distinguished function or role is the occurrence of certain events or effects of interest in the environment. Let us say F stands for such an identified function or role. Intentional agents often have *purposes* to cause certain events or effects in the environment. If agents have a purpose to cause effect F in the environment, and, in order to achieve this purpose, if they use a certain object that causes F, then they may say that the object has the function F. Suppose a theorist wishes to explain a certain effect F in some domain. If she believes that some object O causes the effect F, she may say that O has the role F in the domain. All of these concepts use as their central element the idea of a distinguished effect of interest.

Distinguished Effects (or Functional Predicates) of Interest. The most general version of these is given by a

set of {conditions, effects} where both conditions and effects are specified as predicates or temporal sequences of predicates defined over environmental state variables. The functional predicates or effects are defined purely in terms of environmental variables. They make no reference to the properties of any object that might be introduced into the environment to cause the effects.

Examples. The `buzz` functional predicate. A selected part of the environment has a buzzing sound in it.

The `sawtooth` functional predicate. The voltage between two given terminals in the environment to rise over interval T from 0 to V, then instantaneously drop to zero, and this pattern to be repeated.

Function of an Object. Given a functional effect, how do we relate it to the function of an object? Since the functional predicates and conditions may not make any reference to any part of the structure of the object, we need to describe a *mode of deployment* of the object to make the link between an object and defined effect of interest.

Mode of Deployment. Given an object O, a mode of deployment specifies:

1. How O is to be connected to its environment, i.e., how it is to be configured such that the environment can effect certain selected properties of O and O can effect selected properties of the environment. A typical way to specify this is to define *ports* of different types for O as well as the objects in the environment and describe which ports of O are connected to which ports of the environment.

2. Required external causes on the object, or, more generally, required sequences of interactions between O and external objects. In the case of a device that is to be used by an external user (as opposed to a device to be connected to other devices to make a composite device), the external causes are specified in terms of actions by a user on O.

Scenarios. The sequence of interactions between the user and a system has been called a *scenario* in the literature on interactive systems and Software Engineering [1-4]. Kaindl [6] emphasized that these are *required* interactions, and so scenarios can be viewed as *behavioral requirements*. We call them *interaction requirements* in this paper, and in order to make the notion of a scenario less ambiguous we call it an *interaction scenario*.

Ascribing a function to an object. Given a function F and an object O, and a mode of deployment, M, we can say that F is a function of O, if there is a mode of deployment M such that O under M causes the effects specified in F under associated conditions.

Intended Function and User Purpose. Function as defined above is neutral with respect to whether the effect on the environment is intended, as in the domain of devices, something undesired, as in the effect that a malfunctioning device might have on the environment, or simply a description of a fact, as in scientific descriptions where talk of intentions of nature are to be avoided. The more neutral term such as "role" is used to describe the latter. We elaborate here our earlier discussion on the relation between agents' purposes, roles of objects and functions of devices.

User purpose: a user intends or desires a certain effect in the world under certain conditions. These effects are described using the {conditions, effects} formalism described earlier. Let us say that the user intends the effect F.

Designer task: The task of the designer is as follows. He is given F as part of the requirements and has to:

1. describe an object, i.e., a set of components from some agreed on repertoire of objects and their configuration, and
2. a mode of deployment of the object

such that under the described mode of deployment, the object causes the effects in F. Then a function, defined by F, can be attributed to the object.

A main point here is that what unites the user's purpose, the designer's task and the function of the object is the effect on the environment. The object causes those effects and thus has a function defined in terms of the effects. The user wants those effects, and hence looks for an object which has a function of causing those effects. The designer is tasked with making an object which has the function.

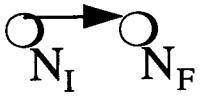
4. FIRQ Specification for ATM

Now we are ready to discuss requirement specification using the ATM example. Kaindl [6] links scenarios (in the sense of behavioral/interaction requirements) with functional requirements, and uses earlier FR work to define the underlying semantics. In this section, we develop this more precisely and elaborate on it. Interaction scenarios are one aspect of the "mode of deployment" that we talked about earlier.

The Environment. Consider an environment composed of a bank with customer account records, and (unspecified

number of) persons some of whom are bank customers (that is, they have bank accounts). We will denote a generic user by U, a generic customer by C, a customer's account by #(C), the balance in the account by B(#(C)), and the cash that a user U has by c(U).

Effect or Functional Predicates for the ATM's `Withdraw-cash($w,$L)` Function. Let us say the bank officials would like a device one of whose functions is to let legitimate customers withdraw cash, up to a limit $L, from their accounts. We define the functional predicates as in the following figure.



We define the effects of interest by defining an initial state N_I and a final state N_F, each with certain properties. We would like the device to cause the transition from N_I to N_F. Using the {conditions, effects} formulation, N_I defines the initial predicates and N_F the final predicates.

Functional (Effects) Predicates
N_I is defined by the predicates: c(U) = $x, B(#(U)) = $y N_F is defined by the predicates c(U) = $x + $w, B((#U)) = $y - $w

One purpose of the bank officials for the ATM can be called `Withdraw_cash(C,$w,$L)`, i.e., they would like their customers to withdraw cash (within the withdrawal limit `$L` set per withdrawal). The purpose is to cause the above predicates to be true under the following conditions:

Conditions
U's purpose is (c(U) = $x + $w) U is a C $w < $y, $L

They would like a device which can cause N_F to be true, given N_I as the initial state and under the conditions above.

The function of the device ATM can also be called `Withdraw_Cash(C,$w,$L)` and defined as the causing of the effects described under above conditions.

The purpose of a user C regarding an ATM can be described at several levels of description.

1. `Get_cash($w)`, where $x is his cash reserve at the initial state and $x + $w is the cash at the final state.
2. `Withdraw_cash($w)`, where the cash reserves at the initial and final states are as in 1 above, but, additionally, imposes conditions on the balance in his account.

`Withdraw_cash($w)` is a special case of `Get_cash($w)`. When the issue is to get some cash (`Get_cash($w)`), a bank customer may choose `Withdraw_cash($w)` and might look for a device which will cause the corresponding N_F to become true[1]. In that case, the ATM might be a possible device, since its functional definition suggests that the ATM can cause N_F to become true. Since there are many other ways to realize `Get_cash($w)` — borrow from someone, steal it, and so on — it is much more appropriate for the user to ascribe the `Withdraw_cash(C,$w)` function to the ATM than the more general function of `Get_cash($w)`.

The main point of the above discussion is to illustrate the central role played by the functional predicates in defining the function abstractly and in relating the function to purposes of agents, users and designers alike. There are small differences in the way we defined the purposes of the bank officials and a user, e.g. — the officials might be more naturally interested than a user in limiting the amount of withdrawal to $L. There are also differences in the way the function of the ATM and the purpose of a user are defined — the function is defined as allowing any customer to withdraw cash, while for a given customer C, his purpose is defined in terms of *his* being able to withdraw cash. These minor differences aside, the functional predicates described in the table are at the heart of descriptions of the purposes of the various intentional agents and the function of the ATM.

Interaction Design Refinement

The function as specified above can be given to the designer as part of the requirements. Let us imagine that either the designer and/or the bank officials engage in some additional interaction design. The product of this

[1] How one matches a goal (in this case `Get_cash($w)`) to a device function (`Withdraw_cash($w)`) is an issue that recurs in the literature on reasoning about function. For example, Umeda, *et al* [9] discuss search for a component that can fulfill a given function. Liver [10] describes an algorithm for incrementally backing off of requirements until a matching function can be found. The issues related to matching are important, but not central to our main points, so we do not discuss this issue further in this paper.

design is going to be certain commitments about the way the device is to achieve its function. In particular, we wish to focus on specifying the required interactions between a user and the device.

Scenarios Emerge from Design by Function Decomposition. As mentioned earlier, the designer's task is to come up with a device and instructions for deploying it so that the function is achieved. Deploying a device involves specifying input and output ports and any user interactions needed at the ports to achieve the function. The function is defined as a set of predicates to be true at a final state (given another set of predicates is true at the initial state). These predicates can often suggest a top-level decomposition of the design task into subtasks. The point of interest to us here is that some of the subtasks may have a solution involving user interaction. Thus, as we move from a function definition which only focuses abstractly on the predicates intended to be true, to a series of refinements and decompositions into subtasks, an interaction scenario starts emerging as well.

In the ATM example, by looking at the function definition, we can identify a number of subtasks that the device has to perform:

1. Give U a means of expressing purpose, "Withdraw $w."
2. Verify U is a C.
3. Verify $w < $L and < B(#(C)).
4. Update B(#(C)) by subtracting $w from it.
5. Dispense $w if 2 and 3 above are satisfied.

We can get some information regarding subtask ordering by examining the preconditions of the subtasks: clearly subtask 1 has to precede subtask 3, and subtasks 2 and 3 have to precede subtasks 4 and 5.

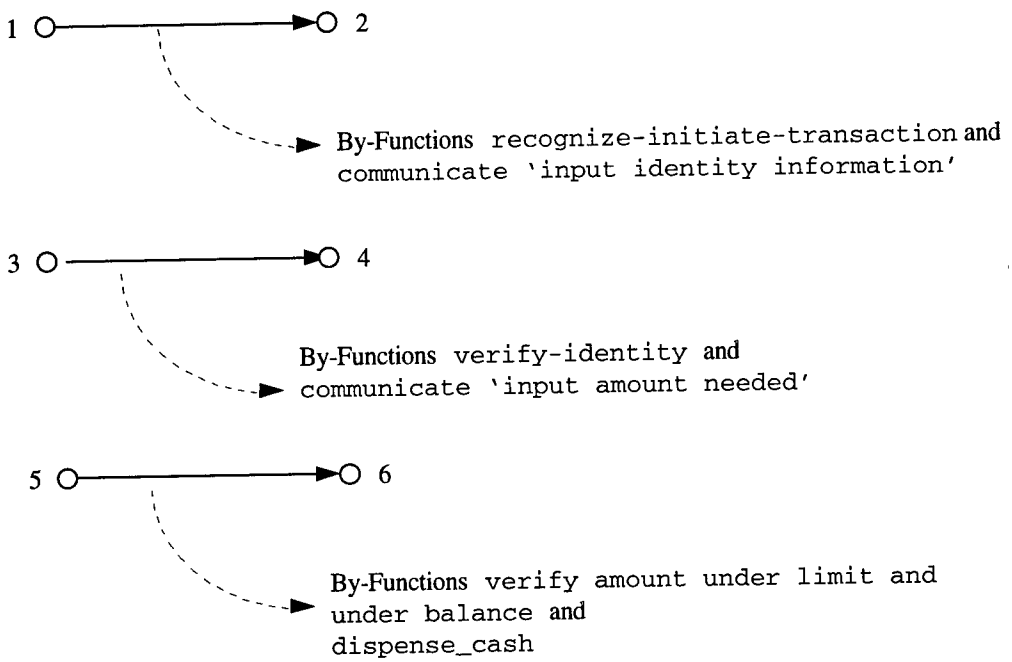
Let us just focus on the user-device interaction. Subtasks 3 and 4 do not require any interaction with the user — they call for interaction with the bank accounts database. Subtasks 1, 2 and 5 together determine the basis for the interaction scenario.

The scenario itself can be abstract [3] — making little commitment to the details of the device structure — or it can be concrete, involving commitments about ports, the places where user-device interaction takes place.

The design task decomposition above immediately suggests an *abstract scenario* of interaction:

1. The user initiates a withdrawal transaction
2. The ATM requests identification
3. The user provides identification
4. The ATM asks for the amount needed
5. The user communicates the amount needed
6. The ATM dispenses cash
7. The user takes the cash.

Each of the items in the sequence above is an *action*, either of the user or of the system. In this representation, we treat actions as the user being in certain states. Actions have effects on things acted upon just like any state of an entity that causally affects a state of another entity. Thus, the abstract scenario is a kind of causal process description. The transitions are one of two types: it either involves a device (ATM) function or a user function. Following the CPD representation, we can annotate the transitions corresponding to the ATM functions as follows.



The transitions 2 to 3, 4 to 5 and 6 to 7 involve user actions. We can, if we like, annotate them using By-"Function <> of User," but we have not done so in the above diagram because our focus is on the functional requirements of the ATM. If the ATM is embedded as a component in a larger system, where the user functions are performed by some other component of the larger system, then of course, it would be quite natural to encode the transitions using the By-Function annotation. In fact, this uniform way of treating user functions and component functions is one of the attractions of the CPD approach to encoding user interactions — users perform certain functions which then enable the device to go into certain states where they then perform other functions.

Each of the functions above can be defined using the functional definition framework described earlier. Some of the functions in the annotations of the transitions can be thought of functions of the components (yet to be designed) of the ATM while others can be attributed to the ATM as a whole. In the course of Requirements Engineering, it is not yet to be defined which components the ATM consists of. However, the ports can be defined where the effect of a function is to be achieved.

For example, consider `dispense_cash($w)`. Let P be a specified port of the ATM. The functional predicate that defines the function is:

Condition: Given $0 at port P
Effect: (ToMake) $w at P

The above abstract scenario, along with the functions that account for the transitions, can serve as part of the requirements specification. The scenario can be much more concrete as well — if, as part of initial interaction design, additional commitments are made, and thus become part of the charge to the designers.

More Concrete Scenarios. Suppose it is decided that users will have a card with their account numbers, they will be assigned a password, and that (overall) the following ports will be available for interaction.

- P_1, for user placing the card
- P_2, for user to input password and desired amount
- P_3, for the ATM to inform user of actions needed
- P_4, for delivery of cash

The scenario can now be written as:

1. The user places card at P_1
2. The ATM displays message at P_3 : "Input password"
3. The user inputs password at P_2
4. The ATM displays message at P_3 : "Input the amount needed"
5. The user inputs the amount needed at P_2
6. The ATM deposits cash at P_4
7. The user takes the cash from P_4.

The transitions can be annotated as before, except that the functions can be much more specific about the ports at which certain functional predicates have to apply.

Why the CPD Works for Representing the Interaction Scenario. Why does the process description formalism developed for explaining how a device works prove useful for specifying user-system interactions? One can view the user plus the device as yet another "device" in which the user herself is a component causally interacting with another component. Thus, user's action states are like the states of any other component. The interaction scenario simply becomes the causal process description for this larger device. While for the presentation above we had attributed the function `Withdraw_Cash(C,$w)` to the ATM device alone, actually the ATM and the user have to cooperate in order to achieve it. Also the user, as a component, has a certain number of functions to achieve, just like any other component such as the ATM. The interaction scenario then becomes a causal process description of how the user-ATM combination works. So, `Withdraw_cash(C,$w)` is really the function of the ATM plus the user. That is why, as we refine the design, the interaction scenario emerges from functional decomposition, by making explicit also the functions required from the user.

How Such a Representation Can Be Used. In earlier work of one of us (see Kaindl [6]), the primary use of functional representation was to define the underlying semantics of the new approach of linking interaction scenarios with functional requirements and purposes. The cleaner and elaborated representation described above should be even more useful in this respect.

When making use of one of the systems available for functional *reasoning*, our approach may also be useful for automated *analysis* of requirements. Requirements models represented in this way can be simulated in order to identify problems or validate the requirements.

5. Concluding Remarks

The contributions of this paper can be viewed from several perspectives. At the simplest level, it shows the use of a definition of function and the causal process description formalism of the FR framework to represent functional requirements for system design. The approach is as useful for software systems as it is for hardware or hybrid systems, since the terms used to describe functions equally apply to all types of systems. The function framework used here helps to see in a unified way how user purposes in using a device, designer intentions and functions of the device are related and arise from certain basic functional predicates defined in terms of environmental variables.

At another level, the work presented can be viewed as a formalism for representing user-system interactions, a topic of substantial interest to the community concerned with the design of interactive systems. Again, there are a number of unifications: the same representational framework that is used for causal process representations in FR is used to represent user-system interactions. The functional annotations for transitions in CPD set a number of design subtasks for the designer.

Still another dimension of interest is the relation between interaction design commitments and refinement of functional requirements. In summary, this paper shows how modeling in the sense of functional representation and reasoning can be usefully applied to Requirements Engineering and Human-Computer Interaction.

Acknowledgments

B. Chandrasekaran's research was supported by ARPA, order no. A714, and monitored by USAF Rome Laboratories, contract F30602-93-C-0243. The authors acknowledge with thanks comments by Dean Allemang that helped improve the paper.

References

[1] Carroll, J. M. ed. *Scenario-Based Design*. New York, NY: John Wiley & Sons, 1995.

[2] Carroll, J. M.; Mack, R. L.; Robertson, S. P;. and Rosson, M. B. "Bindings scenarios to objects of use," *International Journal of Human-Computer Studies*, vol. 41, pp. 243-276, 1994.

[3] Constantine, L. "Essential Modeling: Use Cases for User Interfaces," *ACM Interactions*, vol. 2, pp. 34-46, 1995.

[4] Potts, C.; Takahashi, K.; and Anton, A. I. "Inquiry-based requirements analysis," *IEEE Software*, vol. 11, pp. 21-32, 1994.

[5] Chandrasekaran, B. "Functional representation and causal processes," in *Advances in Computers*, vol. 38, M. C. Yovits, Ed.: Academic Press, 1994, pp. 73-143.

[6] Kaindl, H. "An integration of scenarios with their purposes in task modeling," in *Proceedings of the Symposium on Designing Interactive Systems: Processes, Practices, Methods & Techniques (DIS '95)*, pp. 227-235, Ann Arbor, MI, 1995, ACM.

[7] Liver, B., and Allemang, D. T. "A Functional Representation for Software Reuse and Design, *International Journal of Software Engineering and Knowledge Engineering*, vol. 5, pp. 227-269, 1995.

[8] Chandrasekaran, B. "An explication of function," The Ohio State University, Laboratory for AI Research, Columbus, OH, Draft, 1996.

[9] Umeda, Y.; Tomiyama, T.; and Yoshikawa, H. A design methodology for a self-maintenance machine

based on functional redundancy, in *Design Theory and Methodology DTM 92*, D. L. Taylor and L. A. Stauffer, Ed., American Society of Mechanical Engineers, 1992, pp. 317-324.

[10] Liver, B. Working-around faulty communication procedures using functional models, in *Working Notes on the AAAI-93 Workshop on Reasoning about Function*, 1993, pp.95-101.

Functional Representation of Executable Software Architectures [1]

John Hartman

B. Chandrasekaran

Laboratory for Artificial Intelligence Research
Dept. of Computer and Information Science
The Ohio State University
hartman *or* chandra@cis.ohio-state.edu

December 1, 1995

[1]This work was supported by ARPA, Order No. A714, monitored by USAF Materiel Command Rome Laboratories, Contract F30602-93-C-0243.

Abstract

Software architectures specify how high-level system components interact and behave. Architecture evolution tasks require knowing an architecture's design intentions. Existing architecture description languages (ADL's), however, specify architectures without reference to intentions. We describe the use of Functional Representation to capture understanding of design intentions and their implementation in an architecture. This approach will reduce the cost of designing, evolving, and implementing architectures by improving human communication, and by providing more useful tools and environments. Applications include prototyping, dynamic documentation, design verification, simulation, execution analysis, and other architecture activities.

Chandrasekaran's Functional Representation is used to connect design intentions to an architecture's ADL specifications. The result is a rich, hierarchical explanatory structure which is useful for many purposes. Functional Representation is a theory and language for reasoning about functionality and causal processes in devices. It has been successfully applied to a large variety of tasks and devices, including software. Luckham's Rapide is an executable ADL based on a rule-event execution model. FR-Rapide applies Functional Representation to aid architecture prototyping with Rapide.

An example functional representation captures understanding of how part of the Two-Phase Commit protocol is implemented in a Rapide prototype. Its explanation incorporates understanding in domains such as transaction processing, the X/Open standard, concurrent computing, and distributed computing. The FR is a formal representation which helps humans understand and communicate about the architecture. It also allows understanding to be delivered and exploited by tools and environments.

The value of this approach is demonstrated with an explanation tool which supports Rapide prototyping and other architecture activities which can benefit from captured understanding. Applications and tools include browsing, documentation, debugging, simulation, design verification, and rationale capture.

Contents

List of Figures

List of Tables

Chapter 1

Functional Representation of Architecture

Software architectures specify how high-level system components interact and behave. Architecture evolution tasks require knowing an architecture's design intentions. Existing architecture description languages, however, specify architectures without reference to intentions. We describe the use of Functional Representation to capture understanding of design intentions and their implementation in an architecture.

1.1 Understanding for Architecture Evolution

Architecture is the high-level design of complex software systems. It addresses how large-scale system components interact, independent of implementation and non-architectural details. There is a large movement to study and manipulate systems at the architecture level.[3, 9] *Architecture-based methods* design and evolve systems at this level, with reference to specifications of high-level components and their interactions. These specifications are expressed in architecture description languages (ADL's).

Working at the architectural level has many advantages. For example, starting system design at the architectural level allows important design commitments to be worked out early, independent of less important details. The architecture description then guides detailed implementation, and serves as a specification for analysis, debugging, and documentation. More generally, architecture-based methods use the architecture description to control architecture and system evolution. (Figure 1.1). Even original design can be seen as evolution – understanding and modifying the architecture under construction. Subsequent evolution may involve understanding and modifying an unfamiliar architecture and system. For example, in considering implementations of, or changes to, communications channels, it is necessary to understand the purpose of the channels for achieving top-level system goals. In a distributed transaction

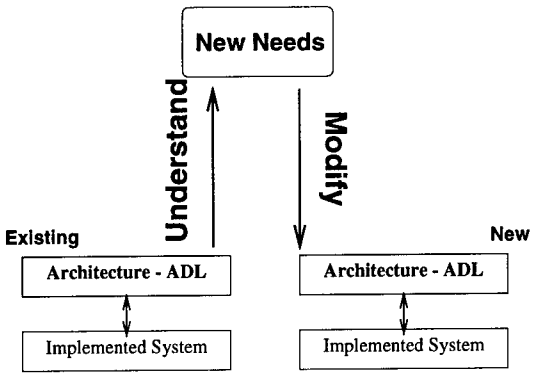


Figure 1.1: Architecture-Based Evolution Processes

processing architecture, transactions may be communicated between applications and resources with particular constraints. To implement or change the system, it is necessary to understand the role of these constraints in relation to top-level design goals.

What is the nature of the understanding needed for architecture evolution tasks? Intuitively, architecture understanding is open-ended and exceedingly

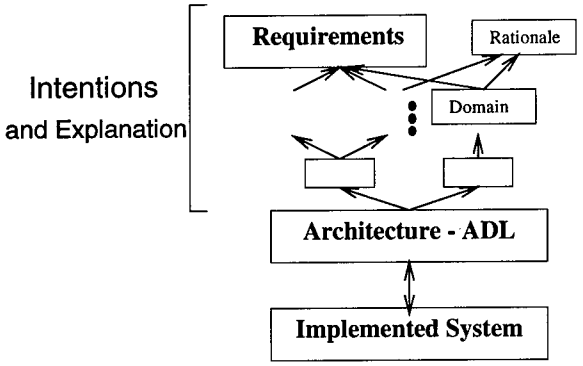


Figure 1.2: Architecture Understanding

complex. It includes assertions about the architecture needed to perform arbitrary architecture tasks. These assertions may be at many abstraction levels, in many descriptive frameworks, and may incorporate knowledge and inference from many domains, including computer science and engineering, mathematics, and the computational problem and its domain. Furthermore, assertions about the architecture are connected together in a complex web of deductions,

hypotheses, dependencies, justifications, definitions etc. (Figure 1.2)

Intentions are assertions about the architecture that designers (and maintainers) may reasonably have. Requirements are intentions about the role of the system in achieving its purpose in the computational problem domain. There are many ancillary and subsidiary intentions, e.g. non-functional requirements, analyses, design commitments, rationales, specifications, refinements, implementations, behavior descriptions etc.

Explanations are assertions and inference links which connect intentions to each other, and which connect intentions to the architecture. Explanations can be seen as the logical structure which answers questions about how intentions are achieved by means of other intentions and the architecture. In practice, intentions and explanations may be mental and/or written, and may have any temporal extent, e.g. fleeting, as needed, or permanent.

1.2 Goal – Capture Causal Understanding

All this says that capturing architecture understanding is akin to capturing full human understanding, which is beyond the scope of this report. We narrow our representation objectives in several ways. First, we start with an architecture specified with an ADL. Our goal is to capture intentions that are relatively close to the architecture, and which do not involve large amounts of problem domain or other information or knowledge. That is, we don't want to go too high in abstraction levels, being content with computational requirements like, "Preserve system consistency regardless of transaction order," rather than domain requirements like, "Retrieve patient data." Similarly, we needn't go lower than the ADL, both because we are supporting architecture-based evolution, and because a more detailed implementation may not exist. Thus the domain of discourse is constrained by the relatively simple kinds of concepts that are present in ADL's.

Secondly, we are only interested in capturing understanding that can be practically and usefully fixed and recorded. This imposes limits on size and complexity. Rather than hopelessly attempting to capture the vast fabric of potential human understanding, we want to record a small subset that is powerfully useful for communication, standardization, and automated assistance. The subset is only useful if it is small enough to be comprehended and applied. Therefore we envision information volume roughly like existing documentation, limited by reading/browsing ability (whether in paper or electronic media), by creation cost, and by other pragmatic considerations.

Finally, we narrow our focus to a particular class of intentions and explanations. We address intentions regarding *causal processes* in the architecture. Temporal quality is an essential characteristic of causal processes. Roughly, therefore, we are concerned with intentions regarding temporal sequences of events and states in the architecture. We are not concerned with intentions and aspects of the architecture that do not have temporal, behavioral quality and consequences. This eliminates much traditional ADL content, e.g. for-

mal relationships such as modularity, interface types, integrity constraints and non-functional properties. This focus requires an architecture which specifies temporal behavior, an *executable architecture*.

Our goal, then, is to capture human causal understanding of architectures in a way which will help people who design, implement, modify and otherwise evolve architectures. Specifically, we will add a layer of explanation connecting intentions to existing ADL's, to assist activities performed with ADL's. The understanding will recorded by a human, using the process, framework, and language to be described. This captured understanding will be useful for further human understanding and communication. It can also be exploited by a large range of tools and environments.

For more motivation, imagine a specific architecture evolution task, e.g. modifying a transaction manager component's behavioral specification. For each such task, it is necessary to know certain intentions, e.g. "The component determines whether transactions should be made permanent," and how and where these intentions are expressed in the architecture. Providing such understanding can save the arbitrarily high effort and expense that the evolver must otherwise expend recreating it. Functional Representation is a well-established means to capture such understanding.

1.3 Functional Representation

Functional Representation (FR) is a theory about understanding devices. It addresses how functional, causal device understanding is represented and used. It has been successfully applied to a large variety of tasks and devices, e.g. explaining failures in a chemical plant, medical diagnosis, and engineering design verification.[2] FR provides a framework, process, and language for capturing understanding of many kinds of devices.

Allemang showed how Functional Representation can capture particular explanations of a program.[1] In his work, an FR is an argument structure which gives a program correctness proof. Allemang formalized this use of Functional Representation as program *functional semantics*, and demonstrated its advantages over traditional programming language semantics for certain kinds of reasoning.

We extend this approach to architectures. Architectures pose additional problems like distribution, concurrency, and weak procedural specification. Furthermore, these and other aspects require more heterogeneous explanation styles and representations. Therefore the need and benefit for representing architecture understanding is at least as great as for simple programs. Furthermore, architectures provide a challenging testbed for Functional Representation, where explanations entail complex and varied description styles, viewpoints, and applications.

Functional Representation is organized around definitions and representations for structure, behavior, and function. *Structure* is the assumed bottom-level description of a device. *Behavior* is temporal, causal change in the device,

particularly changes in its states ("how"). *Function* is an interpretation of the device and its behavior as serving a role in a more abstract context ("what"). There are many–many relationships among possible device structures, behaviors and functions. For example, given functions may have multiple realizations by different possible behaviors and structures.

These definitions are relative to an assumed bottom, structural level. This level is arbitrary. Therefore it is possible for functions to serve as another structural level, providing (abstract) behavior for still more abstract functional description.

The details of the FR representation will be described when we present an architecture FR in Chapter 3. The features which make FR especially suitable for representing intentional, causal architecture understanding, as described above, are:

Functions Formal specifications give functionalities in terms of possible states. They describe abstract intentions or views.

Abstraction Hierarchy Functions are proven abstractions for behavior and explanations at lower levels. Explanation is organized by the hierarchy.

Causal Process Descriptions Behavior is described by state transitions. Transitions are annotated by links giving realization, justification etc.

Components Function is shown to emerge from structure consisting of a particular composition of components. Components are abstract sub-devices which modularize the explanation.

The FR language consists of syntax and semantics for representing understanding in terms of these features.

Applying FR involves: 1) *authoring* a functional representation which represents a particular human understanding of a particular device, and 2) *applications*, in which the functional representation is used to help perform needed tasks. Applications may be manual, where the functional representation is a formalized notation for human communication. Applications may also be automatic, where tools usefully deliver the understanding formalized in the functional representation. For example, FR-based architecture tools can answer questions, guide browsing, generate dynamic, structured documentation, and perform inferences forward or backward in causal chains.

1.4 Functional Representation of Executable Architectures – FR-Rapide

This report describes the use of Functional Representation in architecture evolution. FR represents architectural understanding based on causal description, functional abstractions, and component structure. Representing architectures

in FR allows many applications to exploit understanding using these principles. We are investigating these topics by applying FR to executable architectures specified by David Luckham's Rapide architecture definition language from Stanford.

Rapide is a language for rapidly prototyping, testing, and analyzing executable architectures.[11, 12] The architecture designer specifies the components, connections and constraints of an architecture using Rapide. Components have interfaces by which they can interact with other components when connected by specific connections. Constraints include invariant properties and abstract behaviors that must be satisfied by implementations. Rapide architectures may be executed using abstract behaviors and/or implemented components. The result is a trace, consisting of a partial ordering of dependent events, which summarizes many possible ultimate executions by final implementations.

Rapide is a good representative of ADL's and executable architecture design. It has has been shown to be useful in many architecture evolution tasks. A variety of architectures have been specified, and there is a large body of Rapide code. It has a well-developed specification, environment, and community.

We are therefore investigating application of FR to executable architecture using Rapide. Our approach is tested and demonstrated in FR-Rapide. *FR-Rapide* is a method for capturing and exploiting understanding of Rapide architecture. A Rapide architecture is given. An FR author writes a functional representation expressing his or her understanding of the architecture. The Rapide user uses the FR for subsequent architecture design and evolution, perhaps through assistant tools. Therefore the application of FR to architecture is demonstrated in a method for assisting architecture design and evolution with Rapide.

1.4.1 Authoring

The FR author uses the FR language to represent understanding of the Rapide architecture prototype. A Rapide prototype and an FR for it are illustrated in Figure 1.3.

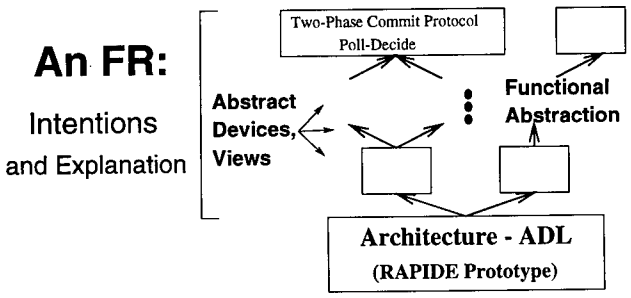


Figure 1.3: An FR Connects Intentions and Architecture

The FR will be constructed for a purpose, e.g. general documentation or supporting a task like protocol evolution. In light of the goal, representing understanding in an FR requires the activities given in Table 1.1. These activities

1. Having or acquiring needed understanding of the architecture,
2. Choosing top-level and intermediate intentions and views to represent,
3. Choosing state languages, formalisms and predicate vocabulary,
4. Writing functional specifications for states at various levels,
5. Forming causal sequences among states at each level
6. Finding functions to form an explanation hierarchy,
7. Modularizing the explanation into abstract devices or components,
8. Validating the explanation.

Table 1.1: FR Authoring

can occur in various orders and iterations. We conservatively called them "representing understanding". In fact they *create understanding* by formalizing intuitions and developing new views, connections, arguments, decompositions etc. Therefore the process has the complex, individualistic, open-ended character of other human comprehension and creative activities. FR authoring activities are described and illustrated in Chapter 3. Here we elaborate each activity to introduce FR and FR-Rapide authoring:

1. Understanding the Architecture Having or acquiring needed understanding of the architecture means the FR author has goals for the FR, and is able to understand the architecture and represent that understanding so the goals can be achieved. If the FR author is the architecture designer, presumably he or she remembers or can recreate his or her design intentions. At the other extreme, an FR author unfamiliar with the architecture may have to recreate (or create) putative intentions by referring to design information sources and/or reverse engineering.

2. Choosing Intentions This understanding is focused around the chosen top-level and intermediate intentions and views to represent. Top-level intentions are the highest needed to satisfy the goals of the FR. For example, suppose the FR is intended to capture understanding of protocols in a transaction processing architecture. Understanding resource access protocols includes understanding how they satisfy requirements such as transaction atomicity and

indivisibility. A particular architecture may be understood to achieve such requirements using a design incorporating the well-known Two-Phase Commit protocol. Two-Phase Commit may then be chosen as a top-level design requirement and intention which is consistent with the goals for the FR. Top-level and intermediate intentions may be essential to all implementations of Two-Phase Commit, e.g. a Poll-Decide procedure which polls Resource Managers and decides when to commit transactions. Intentions may also be architecture-specific, e.g. the design of Poll-Decide in a particular architecture committed to particular components and connections. The top-level and supporting intentions, or views, form the outline of a hierarchical explanation.

3. and 4. Specifying States These intentions must be expressed in appropriate languages. The choice of language depends on FR goals and the intentions' domains of discourse. The languages for top-level intentions will chosen in relation to the goals of the FR. The languages for lower level intentions may be chosen in relation to both both top-level considerations and the kind of explanation desired. In all cases, the languages embody appropriate formalisms and vocabulary.

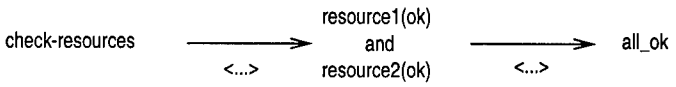
Functional Representation is based on states. Therefore languages are needed which capture possible concrete and abstract states of the architecture. *Concrete states* are sets of values of observable, time-varying architecture properties. *Abstract states* are sets of values of derived or interpreted properties not explicitly present in the architecture. States may be *complete*, completely characterizing the complete architecture, or *partial*, characterizing only some aspects of the architecture. A vocabulary of *predicates* describes sets of states useful in each domain of discourse. Intentions are then written as functional specifications in a language of states and predicates appropriate for the abstraction level and goals of the intentions.

For example, at the architecture level the ADL provides a language for architecture states such as variable values and events, and for intentions, such as the ADL sub-language used to write temporal architectural constraints. Languages for abstraction domains chosen by the FR author may derive from ADL language or may be unrelated. For example, predicates such as "all_ok" or "some_error" can be used to describe abstract states based on semantics of "error" given by an interpretation of concrete architecture states. Other predicates and specifications can convey meaning still further from the ADL and closer to the top-level intentions. For example, "implements_Two-Phase_Commit" is descriptive in the transaction processing domain, and imports complex semantics about what is meant by the Two-Phase Commit protocol, and how its presence can be shown.

5. Forming Causal Sequences Within each level, states are connected in causal sequences to capture the designer's intentions. Besides states, Functional Representation is based on causal transitions between states. An executable architecture undergoes undergoes state changes as it executes. Therefore, it

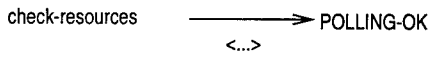
must be understood as a dynamic device, where design intentions are causal sequences of states. FR represents such concrete and abstract behavior and intentions with causal process descriptions. *Causal process descriptions* (CPD's) are state transition diagrams with annotations which describe the transitions. We will focus on the transitions here and on annotations in the following activity.

Forming causal sequences means identifying causal relationships among states and representing them in a state transition diagram. For example, suppose three abstract states were identified in a certain view of the transaction processing architecture: "check-resources", "resource1(ok) and resource2(ok)", and "all_ok". Suppose also they are understood to always occur in a temporal, causal sequence in the order given. Then the following causal process description is written:



The annotations on the causal links are empty or incomplete. The causal relationship has been identified, but the function represented by the links has not been fully specified.

Here is another causal process description in a more abstract domain of discourse:



The state language of this abstraction level shares the "check-resources" predicate with the previous example. It also introduces the new state specification "POLLING-OK".

6. Functions and the Explanation Functions are used to create a hierarchical explanation incorporating and justifying the causal process descriptions. As a result of the previous activities the FR author has created causal process descriptions describing the architecture with various levels of abstraction and views. The CPD's show causal relationships, but lack annotations describing and justifying these relationships.

In FR, *functions* describe abstract functionality. They relate behavior to its role in a more abstract description. Viewed bottom-up, a function abstracts behavior, e.g. a sequence of state transitions, as a single state transition in a more abstract domain. Viewed top-down from the abstract domain, a function justification explains how its role is achieved in terms of more concrete understanding and other justification such as domain knowledge.

Functions also connect the abstraction levels and CPD's into a complete *hierarchical explanation of the architecture.* The explanation is a hierarchy where top-level intentions can be traced through intermediate intentions to base-level implementing structure in the architecture. We say that functions create *functional abstraction* which induces the explanation hierarchy.

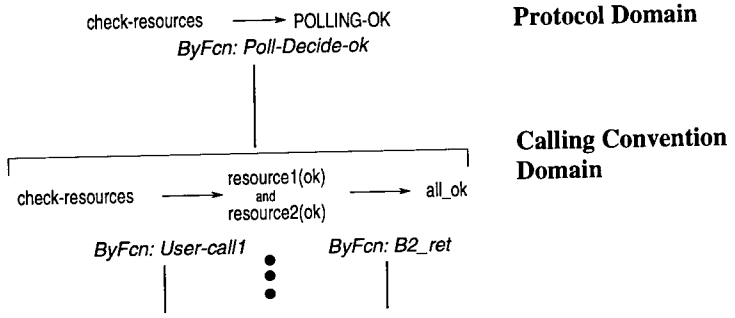


Figure 1.4: Functional Abstraction Creates a Hierarchical Explanation

Figure 1.4 shows part of an explanation where functions complete and connect the CPD's given above. First note that the figure shows the Protocol Domain above the Calling Convention Domain. Each of these abstraction levels or domains of discourse contains particular views or models of the architecture, expressed in particular formalisms and languages. The complete explanation is a hierarchy in which each level is justified and explained only in terms of lower levels.

Next note that function names have been added in "ByFcn" annotations of the CPD links. Functions create the abstraction hierarchy and justify state transitions. Specifically, functions are defined which show how a transition in a higher domain is achieved by behavior in a lower domain. Other kinds of justifications can also be captured by the function, including domain knowledge and definitions. Here the function "Poll-Decide-ok" expresses functionality in the Protocol Domain whereby the state "POLLING-OK" follows the state "check-resources". This transition is explained by the indicated transitions in the lower domain, and by a definition relating the "all_ok" state in the lower domain to "POLLING-OK". These semantics are captured in the justification of the "Poll-Decide-ok"function. Similarly, the functions "User-call1" and "B2_ret" capture functionality creating the indicated state transitions in the Calling Convention Domain. These functions are described in terms of lower views, behaviors and domain principles.

7. Forming FR Components The FR author may also decompose the explanation into abstract devices or components. The activities above combine all of the essential elements of the FR into an explanation of the architecture. Such explanations, however, may be unmanageable because of their size and complexity. Furthermore, they may have lost information about modular groupings in the architecture.

FR contains a mechanism for dividing the explanation into comfortable pieces, and for modeling the architecture as interacting components, at all lev-

els of abstraction. *FR components*, or abstract sub-devices, modularize CPD's within an abstraction level. They can be viewed as named boxes encompassing sets of states. The FR author can define components as desired, e.g. to correspond with domain concepts, understood relationships, or base architecture structure (Figure 1.5). They are primarily grouping constructs. They can, how-

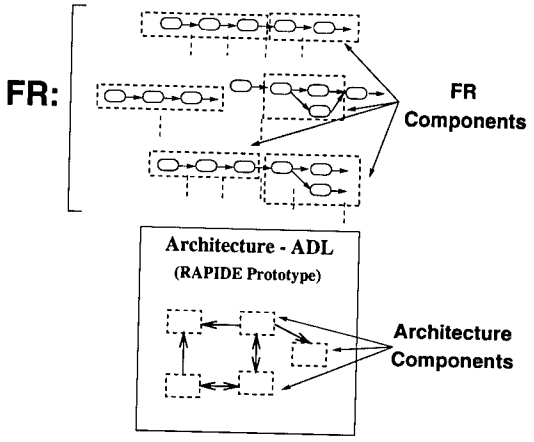


Figure 1.5: FR Components – Abstract Sub-Devices

ever, serve as attachment points for additional semantics. Components allow explanation in terms of interacting components as well as states.

For example, consider the base architecture components, e.g. Resource Managers in a distributed transaction processing architecture. These may or may not be captured as FR components. The FR author is free to create additional FR components, and/or project the architecture components upwards as components in higher abstraction levels. This is possible because the ADL specifies components as units subject to connection, whereas FR components are state-based. Therefore FR components can form abstract sub-devices at all levels of abstraction. Close to the base level, they can divide architecture components based on understanding sub-processes within the components.

Components can also unite material from separate architectural components, thereby representing understanding in a view that recognizes functional relatedness in separate architecture components. This is analogous to logical components in mechanical domains. A manufacturer, for example, may describe a car as containing frame and body components. If we are representing understanding of the car's crash behavior, we may split and lump the base components, e.g. to give a front impact absorption component consisting of the frame and body components which absorb energy in a front collision.

8. Validation Finally, the FR author and others must ensure the validity of the FR for its intended purpose. This can be done with various methods

for various degrees of required rigor. Our approach is independent of the rigor desired, and does not guarantee correctness by any objective measure. That is to say, FR captures a person's understanding. The understanding may be formal or informal, correct or incorrect. FR enforces neither formality nor correctness. FR does, however, formalize the explanation structure. This can be checked for syntactic correctness and consistency.

At one extreme, the architecture may be understood as a formal correctness proof. FR then provides the skeleton syntax in which the proof is written and organized. Validation consists of checking the proof like any other program correctness or mathematical proof, under appropriate domain models and semantics.

At the other extreme, the architecture may be understood as informal, natural language description, like written and/or unwritten documentation and description. In this case, FR can be seen as structuring the description with the formalized concepts and relationships of causality, functional abstraction, states, components etc. Validation then consists of inspecting the FR both for understanding content and usage of the FR language and ontology.

1.4.2 Applications

Suppose we have an FR, capturing an understanding of an architecture, created as described above. What is it good for?

First, it already has been good for creating and systematizing understanding. The extant FR is a notation which records and recalls this understanding for the author. Similarly, the FR is a notation for communication between the author and others. It serves the role of documentation, but with added formalized structure and conventions. Finally, the FR allows captured understanding to be manipulated, delivered, and exploited by automated tools and environments.

In all cases, we answer the question, "What is the FR good for?", with the principle:

Understanding is as understanding does.

Captured understanding has value and application where it makes it easier to perform a specific architecture evolution task. Many tasks require understanding that can obviously be captured in FR's. The value of a particular FR for a task or tasks is an empirical question. Some forms of captured understanding are clearly useful for a wide range of tasks, e.g. documentation.

In assessing the value of FR for architecture evolution tasks we distinguish between, 1) the value of FR content that could exist independent of the FR framework, e.g. isolated architecture models and intentions, 2) the value added by the FR framework, and 3) the possible synergistic value of combining 1) and 2). We claim there is value from 2) and 3) that more than justifies the cost of FR creation, and that this benefit/cost can be demonstrated empirically. However, such empirical studies are beyond the scope of this report. Here we describe how manual and automatic applications benefit from 2), the value added by the FR framework.

Answers to Questions

The FR framework creates certain kinds of explanations, and makes explicit certain states and relationships. For example, it encourages a hierarchically structured explanation in successively more abstract alternative views or domains. States and causal sequences among states are made explicit. Functional abstraction is made explicit. Abstract components control complexity and encapsulate domain concepts. Therefore the FR framework benefits tasks that require knowing such states and relationships.

More specifically,

The FR makes it easy to answer certain important kinds of questions.

For example, "What?" questions can be easily answered by following functional abstractions upwards, and "How?" questions can be answered by tracing CPD's, following function annotations downward when more detail is desired. For example, here is a dialog produced from the explicit relationships in an FR for an architecture containing the Two-Phase Commit Protocol. X/Open-PD6 is a CPD abstractly describing the Poll-Decide procedure in the Two-Phase Commit Protocol Domain.

```
Q> What does X/Open-PD6 do?

A: Poll-Decide for Two-Phase Commit protocol...

Q> How is Poll-Decide implemented?

A: Poll-Decide-ok is followed by Commit,
   or Poll-Decide-error is followed by Rollback.
...

Q> How does B3_Pattern cause B3-some_error?

A: Pattern semantics of Rapide behavior rule B3.
```

In Chapter 4 we catalog the questions that may easily answered by material made explicit by FR, and describe procedures and semantics for answering such questions from the FR. Note well that the ability to answer certain questions easily is a benefit regardless of whether the FR is being read as a notation or being interpreted by an automatic tool, e.g. a question answerer or browser.

Answers Useful for Architecture Evolution

The ability to easily answer certain questions benefits many specific architecture evolution tasks, and tools for those tasks. Furthermore, it is a capability which enables more complex forms of manual and automatic inference. For example, consider the architecture evolution task of algorithm/component replacement. Suppose, for example, it is desired to replace architecture pieces performing a particular function. The FR can be used to: 1) identify the function by name

or description, 2) locate architecture parts implementing the functions, and 3) locate architectural and logical dependencies that must be respected by the replacement.

As another example, consider Rapide debugging or design verification from execution traces. Such debugging requires: 1) identifying an execution anomaly, 2) finding the architecture part creating the incorrect behavior, and 3) correcting the architecture. Because FR is state based, an FR for an architecture's intended design provides state descriptions which may be compared to states reached in execution and recorded in the Rapide trace. Execution anomalies may therefore be identified. FR goes beyond Rapide's existing constraint checking mechanism because it captures specifications, and can interpret traces, in terms of abstract states and behavior as well as the bare events and constraints used by Rapide. Similarly, because the FR captures the architecture at various levels and views, it is possible that one or more of them will provide an appropriate domain for understanding and correcting the bug.

1.5 Plan of Report

The remainder of this report describes writing and using functional representations of executable architectures. Particularly, we describe and illustrate FR-Rapide, and discuss its applications and limitations.

Chapter 2 introduces Rapide and the X/Open reference architecture used as our example. In Chapter 3 we describe creating a functional representation for part of X/Open. The example is used to introduce elements of FR-Rapide including state modeling, architecture and FR components, causal process descriptions, functional abstraction, and domain/view hierarchies.

Chapter 4 describes the value and applications of architecture FR's to easily answer certain questions. We give a catalog of question types which may be answered, procedures for answering them, and discuss the design of a practical question answering explanation tool. Question answering is related to FR semantics. An explanation tool is described which can support Rapide prototyping and other architecture activities.

Chapter 5 evaluates and discusses the limitations, generality, benefits, and prospects for applying FR to architecture-based software engineering. We first introduce issues and a basis for evaluation. Then we evaluate FR-Rapide with respect these criteria at various levels of generality. Applications and tools such as debugging, simulation and rationale capture are discussed. We conclude by reviewing our contributions and suggesting further work and implications.

Chapter 2

Rapide Executable Architecture

This chapter introduces Rapide and the X/Open architecture used as our example. It concludes by discussing Rapide and X/Open architecture understanding needs and representations.

2.1 Rapide - Rapid Architecture Prototyping

Rapide is an architecture description language designed for prototyping architectures of distributed systems.[11, 12] Such systems are complex assemblies of many components. The distributed components operate and interact concurrently. They exchange information across specific communications channels. There are complex timing requirements and constraints.

Rapide provides a language and environment for constructing prototype architectures. Rapide views the architecture as the plan which drives system design, prototyping, development, and validation. Specific emphases of Rapide include: 1) architecture definition that is executable, for early simulation and testing; 2) an execution model that summarizes distributed, concurrent behavior and timing; 3) formal constraints and mappings for architecture definition and comparison; 4) scalability for large industrial systems.

Rapide and its architectures provide a framework for architecture and system design and evolution. Architectures are constructed using the Rapide object-oriented language. The language supports architecture construction with features for describing architecture components (interfaces) and connections. There are also features for specifying behaviors and constraints. The language provides a type system which is used to create the objects which form the architecture prototype.

Once constructed, Rapide architectures are used to test, simulate, validate and otherwise analyze the architecture. Architectures are executed to study their behavior. They are executed by an interpreter, according to the Rapide

execution model, which creates architectural events. Events and their dependencies are recorded in partially ordered set of events called posets. Each poset summarizes many possible system executions of the kind that are recorded in traditional event simulation linear traces. Posets are used to compare architecture behavior to desired behavior, including the behavior of other architectures. The architecture can also be analyzed by runtime checks against formal constraints, and by static formal analysis and verification.

In summary, Rapide architectures are formal, dynamic devices with parts executing concurrently. Functional Representation is designed to capture understanding of such devices.

2.2 Rapide Overview

A Rapide architecture consists of interfaces of modules, connections which describe communications between the modules, and constraints which specify correctness conditions. Interfaces and connections comprise the architecture in the

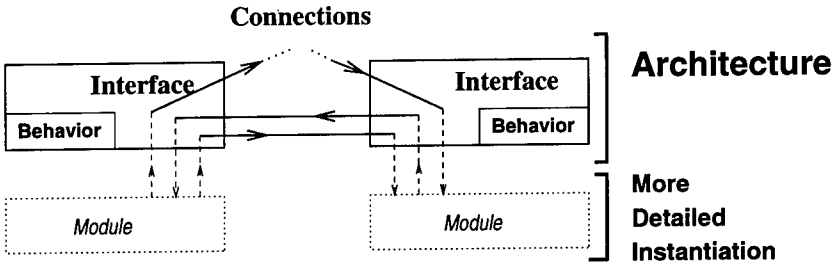


Figure 2.1: Rapide Architecture Terminology

top of Figure 2.1. Possible implementing modules are not part of the architecture, as shown in the lower part of the figure.

Interfaces are the basic architecture components.[1] Interfaces define the features provided to other parts of the architecture, and which must be implemented by modules. Modules are implementations of the interface. The architecture is instantiated when particular modules are assigned to the interfaces. Modules may be executable modules forming an implemented system. Modules may also be other architectures, creating a hierarchical architecture definition.

An interface may contain an *abstract behavior* specifying the behavior required by every implementing module. Many possible modules and kinds of modules can implement a given interface. When no module is present, the

[1]Rapide uses "component" for an interface-module pair, and "interface" for what is usually called a component in the architecture literature. We will continue to use the traditional architecture terminology, where "component" or "architectural component" is the basic architectural unit. We will sometimes use the Rapide terminology in Rapide-specific contexts.

abstract behavior is used in simulation to give the module's effects. Abstract behaviors are given by *executable reactive rules*. These rules recognize situations in the execution and create new events in response.

Connections specify communication between interfaces. They are also defined using executable reactive rules. Connection rules create communication by recognizing output events at interfaces and creating corresponding input events at the connected interfaces. This communication may be asynchronous, or it may be synchronized by a clock. Using rules for connections is an important difference between Rapide and most ADL's, which specify connections with simple static syntax. Using rules for connections allows information passing across connections to treated and recorded by the interpreter in the same manner as other behavior.

The *rule-event-poset execution model* supports simulation and tracing of distributed, concurrent architectures with dynamic structure. *Events* are the indivisible primitives in Rapide executions. They correspond to things that happen in the architecture at run-time, e.g. calling a function, passing a message, changing a memory, or executing a program step. *Posets* are partially ordered sets of events which give the causal history of events and their timing. A poset contains a partial ordering of events connected by their causal dependencies on other events. Causal dependencies are necessary conditions that must precede an event, e.g. an event precondition of the rule generating the new event.

Rapide *processes* are the primitive threads of control. Processes observe the architecture events and the growing poset. When a triggering pattern is recognized, a process generates new events. The new events and their causal dependencies are added to the poset history. Rapide processes may be abstract interface behaviors, connections, or processes in instantiated modules.

The interpreter controls execution in indivisible steps. Within a step, processes observe the current events. They may then create new events. After all rules have observed current events and posted new events, the process repeats. If an event is used by a process to trigger event creation, it may never be used again by that process. Note that the execution steps are based on the cycle of rule interpretation, not fixed periods of time. Thus Rapide execution is dependency-based rather than time-based. This is an important difference between Rapide and most event simulators, which are time-based. Rapide can introduce time as needed with clocks which simulate time by creating events corresponding to specific time intervals.

Formal constraints specify correctness conditions, e.g. communication ordering or data integrity relations. They are attached to interfaces and connections. These constraints are checked at run-time. Constraints can be seen as secondary to the primary functional specification given by abstract interface behaviors and connections. In other words, the behavior and connection rules completely describe the computation. Constraints are specifications for parts of the computation, and have a more abstract, declarative character than the rules.

Note that abstract interface behaviors can have either role. When no module is present, the abstract interface behavior is executed to create the simulation.

When a module is supplied for the interface, the module's processes are used for execution instead of the interface's abstract behavior. The interface's abstract behavior then becomes a constraint. Module generated behavior is checked against this constraint.

2.3 X/Open Architecture

We investigated FR-Rapide using the X/Open Reference Architecture. Kenney developed a Rapide prototype for this architecture[8], and it is the main example in [11].

X/Open is a standard for distributed transaction processing (DTP). The standard defines system component interfaces and sequences of interactions between system components. System components may be applications programs, e.g. a billing system; resource managers for resources, e.g. databases; and transaction managers, which mediate between applications programs and resource managers. The purpose of the standard is to create a standard *reference architecture* for developing and interchanging various distributed transaction processing systems and sub-systems. The reference architecture defines a family of *local instance architectures* which are specialized in aspects like the number and nature of resources and managers. Each local instance architecture, in turn, defines a family of implemented systems.

The X/Open standard consists of over 900 pages of informal English text, with system component interfaces given in C. It describes 1) the interfaces, 2) ways of connecting components which satisfy the standard, and 3) protocols or calling sequences for using the interfaces.

Besides promoting open systems, the standard is intended to ensure certain kinds of correctness in conforming systems. The standard particularly emphasizes protocols which guarantee transaction atomicity. *Atomicity* means that a transaction either executes completely or has no effect. It is ensured by calling sequences which implement the well-known Two-Phase Commit Protocol.

Figure 2.2 shows the X/Open local instance architecture we will use. The boxes are interfaces defined for an application program (AP), a transaction manager (TM) and two resource managers (RM1 and RM2). The arrows are bundles of connections between interfaces called services. Each *service* contains connections between specific functions of the interfaces.

Imagine that the architecture is implemented in a bank. The application program runs on automatic teller machines. It performs transactions involving the bank's checking account and savings account databases, each accessed through a resource manager. The transaction manager mediates between the application program and resource managers as follows: The AP tells the TM it wants to perform a transaction. The TM initializes the RM's and returns a transaction identifier (xid) to the AP. The AP then makes requests and receives results directly from the RM's. When the transaction is completed the AP asks the TM to finalize it. To do this, the TM polls both of the RM's for their approval. If both RM's approve, the TM decides to commit the transaction and

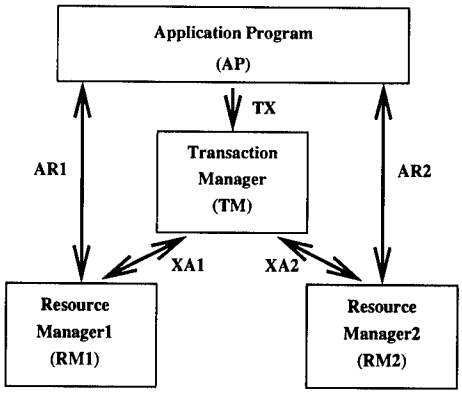


Figure 2.2: An X/Open Architecture

tells the RM's to do so. If they signal success, the TM tells the AP that the transaction is final and usable. If the transaction cannot be committed, e.g. because an RM says it would place a database in an inconsistent state, the TM tells the RM's and the AP to rollback the transaction and start over. Each of these operations is described by particular functions in the interfaces. The functions each have defined connections, which are contained in the services shown.

This scenario and the architecture protocols incorporate the *Two-Phase Commit Protocol*. Two-Phase Commit assures that transactions will be atomic, i.e. they will be completed and committed correctly, or they will be aborted and rolled back. The complete proof that the architecture implements Two-Phase Commit and guarantees atomicity is complex, involving many aspects of the architecture and standard. For example, one necessary property is *coordination*, where the resource manager must get approval from all of the resource managers.

We will focus on *Poll-Decide*, which is a necessary part of Two-Phase Commit and the atomicity guarantee. Poll-Decide is the procedure by which the TM polls the RM's for their approval or disapproval of the transaction (*Polling Phase*), decides to commit or rollback the transaction (*Decision Phase*), and causes the necessary actions. We will further narrow our focus to the connections and abstract behaviors involved in Poll-Decide. See [11] and [8] for more discussion of the interface definitions, services, constraints, and posets.

2.4 Poll-Decide Definition and Behavior

Recall Kenney created an architecture for a version of the X/Open Reference architecture, just as every Rapide author describes an architecture in the Rapide language. We will use it to further describe Rapide, and as the FR-Rapide

example in subsequent chapters. Figure 2.3 summarizes the part of the architecture which describes the connections and abstract behaviors of Poll-Decide.

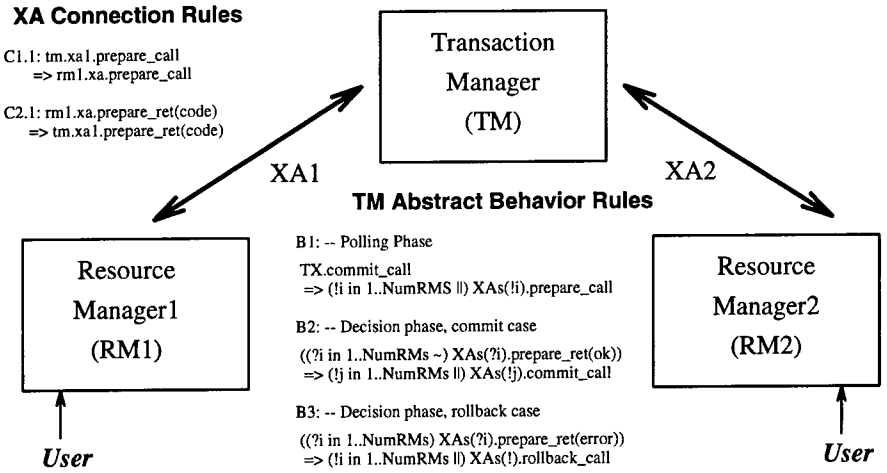


Figure 2.3: X/Open Poll-Decide

In the figure we see the XA connection rules which define connections in the XA1 service between TM and RM1. There are similar rules defining connections in the XA2 service between TM and RM2, but they are not shown. The figure also shows abstract behavior rules defining the Poll-Decide behavior in the Transaction Manager (TM) interface. Since this particular architecture was created with two resource managers, Rapide variable NumRMs used in the rules has the value 2. Other Rapide rule syntax should be more or less obvious from our description below.

The actual Rapide code is more complicated and obscure than Figure 2.3. We do not include the complete Rapide code in this report because it is voluminous and difficult to interpret without detailed knowledge of Rapide. Figure 2.4 shows part of the X/Open Rapide code to give a feel for its syntax and other characteristics.

In the Rapide code there are various levels of type definitions and instantiations for all of the objects involved. Connections are described in detail at both ends. Services are defined at both ends for each communicating pair of components, with reference to all of the bundled connections. Components are defined with reference to services. Constraints and abstract behaviors are attached at various places. Finally, the entire architecture is instantiated from an architecture generator type. Much of the language syntax is designed for rapid architecture configuration and change within families. The resulting complexity makes it especially hard to understand the architecture from the Rapide code.

```
type XA_Service is interface
public
   type return_code is enum
      ax_ok, xa_error
     end;
   action prepare_call(x : xid);
   action commit_call(x : xid);
   action rollback_call(x : xid);
extern
   action prepare_ret(x : xid, rc : return_code);
   action commit_ret(x : xid, rc : return_code);
   action rollback_ret(x : xid, rc : return_code);
...
end XA_Service;

type Transaction_Manager(NumRMs : Integer) is interface
public TX : service TX_Service;
extern XAs : service(1..NumRMs) XA_Service;
behavior
 TX.commit_call
  => (!i in 1..NumRMs ||) XAs(!i).prepare_call;
 ((?i in 1..NumRMs ~) XAs(?i).prepare_ret(ok))
 => (!j in 1..NumRMs ||) XAs(!j).commit_call;
 ((?i in 1..NumRMs) XAs(?i).prepare_ret(error))
 => (!i in 1..NumRMs ||) XAs(!).rollback_call;
...
end Transaction_Manager;

architecture X/Open_Architecture(NumRMs:Integer)
   return X/Open is
   AP : Application_Program(NumRMs);
   TM : Transaction_Manager(NumRMs);
   RMs: array(Integer) of Resource_Manager;
connect
   AP.TX to TM.TX;
   for i:integer in 1..NumRMs generate
      TM.XAs(i) to RMs[i].XA;
      AP.AR(i)  to RMs[i].AR;
...
end architecture X/Open_Architecture;
```

Figure 2.4: Sample X/Open Rapide Code

2.4.1 Simulation

The architecture is analyzed by simulation. Simulation means the architecture is made to execute in a way that simulates the execution of a final implemented system. Of course the architecture only describes certain high-level aspects of the final implemented system, so the simulation will only capture behavior at the architecture level, e.g. message passing and gross behavior such as initiating and completing generalized transactions. Furthermore, the user will design and direct the simulation in order to investigate certain aspects of the architecture, e.g. message timing or protocol correctness.

Events are the primitive elements in Rapide executions. Events are are represented by character strings. For example, `tm.tx.commit_call(xid)` is an event which models the TM receiving a procedure call from the AP which was communicated across the TX service. `xid` is the transaction identifier which is present in most events, but which we will subsequently omit for brevity.

When the architecture is created, it starts executing using initialization behaviors designed to simulate the transaction processing that is being tested. Suppose the user is investigating the architecture's behavior for a single transaction. Let us assume that the simulation has run to the point where the AP has initialized a transaction and transferred all needed data with the RM's. The AP then requests transaction completion, and TM receives the `tm.tx.commit_call` event.

This event matches Rule B1 in TM, initiating the Polling Phase. Rule B1 generates events `tm.xa1.prepare_call` and `tm.xa2.prepare_call`, corresponding to TM issuing calls to the RM's. `tm.xa1.prepare_call` is recognized by Rule C1.1 defining a connection in XA1. This rule creates the event `rm1.xa.prepare_call`, corresponding to the call from TM being received at RM1 across a connection in XA1. A similar rule causes a similar event corresponding to another call from TM being received at RM2. Calling and subsequent execution in RM1 and RM2 is modeled as proceeding concurrently and asynchronously.

The RM's do not have detailed behavior descriptions. Furthermore, for testing the architecture we don't care about detailed database operation. We merely want to test the architecture's response to transaction approval or disapproval from the RM's. Therefore, in the current architecture the user tells each RM whether to signal approval or disapproval. Let us suppose the user wants to simulate approval by both resource managers. The events `rm1.xa.prepare_ret(ok)` and `rm2.xa.prepare_ret(ok)` are therefore generated, corresponding to the RM's returning from the above calls. In the case of RM1 and XA1, Connection Rule C2.1 responds to this event and generates the event `tm.xa1.prepare_ret(ok)`, corresponding to the return being received at the TM. Similarly, the event `tm.xa2.prepare_ret(ok)` is generated, recording TM receiving an approval return from RM2.

The forking into concurrent communication and execution in the RM's now ends. The only abstract behavior that can next execute is B2, and it must wait for both `prepare_ret(ok)` events to be posted. When this occurs, the

concurrent forks are merged into a single control thread in TM. Satisfaction of B2's preconditions causes TM to enter the Decision Phase. Since both RM's approve, TM decides to commit the transaction. This is described by the consequent of rule B2, which creates the events `rm.xa1.commit_call` and `rm.xa2.commit_call`. These events then cause the actions needed to finalize the transaction with the RM's and the AP. Rules for this behavior are not shown.

If either of the RM's had disapproved, a `prepare_ret(error)` return event would be present instead of two `prepare_ret(ok)` events. Then the concurrent fork would be merged by TM behavior rule B3, also initiating the Decision Phase in TM. In this case, TM would decide to rollback the transaction, and B3's consequents would create the events `rm.xa1.rollback_call` and `rm.xa2.rollback_call`.

2.4.2 Causal History Poset

Figure 2.5 is the poset created by the interpreter which records the execution described above. The ovals contain the events described. Following the initiating

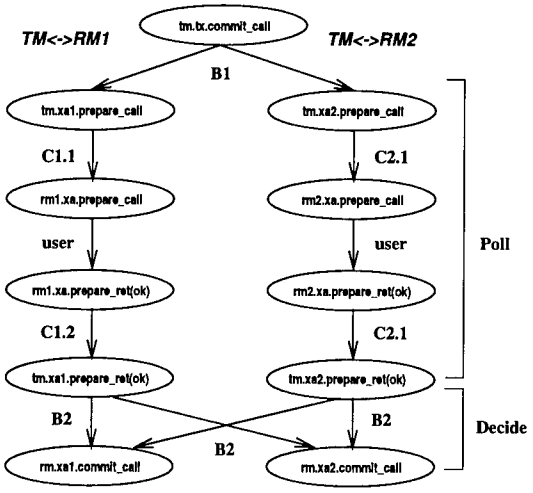


Figure 2.5: Poll-Decide Causal History Poset

event at the top, events involving RM1 are on the left and events involving RM2 are on the right.

Arrows show the dependency partial ordering relationship in the poset. An event at the head of an arrow is immediately dependent on an event at the tail of an arrow. In other words, events at the tails of arrows occur before events at heads of arrows, and potentially cause them. In this example, the dependencies are from the connection rules, behavior rules, and interactions shown. The interpreter recorded the dependencies in the poset when it executed the rules.

This simulation and poset reveal three important characteristics of Rapide and Rapide execution. First, note that the execution is concrete and precise *at the architecture level*. It describes the behavior of the architecture as well as possible, using all of the description present in the architecture definition. Furthermore, the execution is a particular behavior of the architecture, responding to particular input and direction. Execution is not "symbolic" or abstract at the architecture level.

Second, the execution is abstract in comparison to more detailed possible instantiations, especially ultimate implemented systems. Adding more implementation makes more details possible and necessary in concrete executions. However, this added detail is non-architectural. The execution at the architectural level provides an abstract behavior and specification which corresponding detailed executions must conform to. Every implementation must have steps corresponding the architectural events and dependencies. For example, the commit decision must be made based on previous approval responses from the resource managers, regardless of how the resource managers and decision phase are implemented.

Third, the execution is abstract with respect to possible detailed histories, even at the architecture level. The execution model is based on explicit dependencies rather than time. Since execution is concurrent, different actual histories could occur under the same dependencies. The poset summarizes all possible event histories that could occur in linear time. For example, the events in the dependent sequence involving RM1, `tm.xa1.prepare_call` → `rm1.xa.prepare_call` → `rm1.xa.prepare_ret(ok)` → `tm.xa1.prepare_ret(ok)`, on the left side of Figure 2.5 must occur in the order given. However, they can be in any relation to the similar concurrent events involving RM2 on the right side of the figure. That is, the RM1 events can occur before, after, or in arbitrary temporal interleaving with the RM2 events.

An important feature of Rapide is the poset's ability to capture the necessary dependencies and ignore unnecessary temporal detail. In contrast, other event-based simulators create a linear history trace or total temporal ordering. Such a linear trace captures scheduling artifacts as well a necessary dependencies. A Rapide poset subsumes all of the linear histories and traces which could be produced by time-based execution and simulation.

2.5 Understanding the Rapide X/Open Architecture

Using this background, we now consider Rapide and X/Open architecture understanding under the goals given in Chapter 1. First we discuss the kinds of things that must be understood about Rapide architectures using the X/Open example. Then we describe how this understanding can be represented in relation to Rapide.

2.5.1 Design Intentions

Our goal is to capture human causal understanding of architectures in a way which will help people work with architectures. Specifically, we seek to add a layer of explanation connecting design intentions to Rapide architectures, to assist activities performed with Rapide architectures.

Some of the intentions which are realized in the Rapide X/Open Poll-Decide architecture above are listed in Table 2.1.

1. Perform transactions
2. Ensure atomicity
3. Implement Two-Phase Commit
4. Implement Polling
5. Poll RM's concurrently
6. Define distributed TM and RM components
7. Define connections between distributed TM and RM components
8. Pass specific messages between the TM and the RM's, e.g. TM requests approval or disapproval from RM1
9. Control Poll-Decide in the TM component
10. Interpret specific Rapide events, e.g. interpret `xa.prepare_ret(<code>)` events to determine if all RM's approve
11. Fork into concurrent processes in each RM
12. Get all results from concurrent polling before deciding
13. Define specific calling interfaces, e.g. parameter passing in the polling call from TM to RM1
14. Generate specific Rapide events and dependencies, e.g. generate the `tm.xa1.prepare_call` event dependent on the `tm.tx.commit_call` event

Table 2.1: X/Open Poll-Decide Intentions

2.5.2 Representation Issues

Here are some observations about these needed understandings and intentions:

1. None of these intentions is explicitly present in the Rapide code. Each requires interpretation of the Rapide code in domains beyond Rapide syntax and semantics.

2. Intentions are achieved with various amounts of Rapide code. Some of the first intentions involve the entire architecture, whereas some of the other intentions involve involve small elements like single rules.

3. Intentions do not necessarily correspond to single Rapide syntactic categories.

4. Intentions involve different domains of discourse. Domains used include distributed transaction processing, transaction atomicity and correctness, the X/Open standard, C calls, Two-Phase Commit, distributed computing, concurrent computing, the particular instance architecture with two RM's, and the goals of simulation testing.

5. Different domains must be described with different languages, formalisms, models of computing etc. These languages involve abstract states of the architecture.

6. Domains may be seen as views of the architecture which emphasize certain aspects and ignore others.

7. Intentions have varying temporal extent. They may refer to everything that happens, or to just things that happen during a particular time period.

8. Intentions have varying temporal quality. They may describe temporal sequences ("procedural") or temporal invariants ("declarative").

9. Intentions need not be temporal or causal. They may refer to time-invariant, non-temporal characteristics of the architecture, e.g. "Is callable by the AP and the TM."

10. For given abstraction levels and temporal quality, intentions vary in how completely they describe the architecture. They may involve the entire functionality at that level of description, or they may refer to some aspect or property of the architecture's functionality.

11. The domains and intentions may be roughly ordered based on distance from Rapide and closeness to requirements. The given list follows such an ordering. However, the domains and intentions also have orthogonal dimensions.

12. Given intentions involve complex combinations of domains.

13. Intentions can often be best understood in terms of other abstract domains, not the base Rapide architecture.

14. It may be possible to order intentions so they are explained in terms of lower intentions, even if there is no monotonic ordering of domains.

15. The realization of intentions can be seen in the poset history as well as in the static architecture description.

16. Poll-Decide intentions directly pertain to single transactions. Our understanding, and the simulation, use *single transaction abstraction*, a view where the architecture is analyzed by considering a single transaction independent of other transactions that might be taking place concurrently. This is supported by understanding that each transaction has a unique transaction identifier.

17. There is a mapping between the poset history and the Rapide syntax which produces it. The also reflects other factors such as data and user interaction.

18. Intentions can be explained using logical groupings which differ from the architectural groupings of connections and components.

19. Rapide constraints capture intentions, at an abstraction level close to Rapide. However, the constraints may not be consistent with the executable part of the architecture or possible executions. This may be because the architecture is incorrect with respect to the constraints, or because the execution has unanticipated input or interactions.

20. The Rapide architecture also reflects non-functional intentions such as clarity and execution efficiency.

These complex needs and issues must be addressed by every approach to representing understanding of Rapide architectures and X/Open.

2.6 Summary

Rapide specifies an architecture at a level that is appropriate for definition and execution. The architecture, like a program, can be executed to produce a trace. Unlike programs, the architecture may contain constraints specifying certain required behaviors.

Tasks involving Rapide architectures require understanding that is not present in the Rapide code. We described such understanding by giving intentions present in the X/Open architecture. Issues, dimensions and criteria for representing these intentions were given. Representing understanding of Rapide architectures differs from representing understanding of simple programs because of Rapide characteristics including the rule-event-poset execution model, distribution, concurrency, and the presence of constraints. The remainder of the report describes the use of Functional Representation to represent and exploit such understanding.

Chapter 3

Functional Representation of the X/Open Architecture

In this chapter we describe a functional representation for the X/Open Poll-Decide architecture. We discuss authoring decisions, the complete hierarchy, the top level, and the bottom level. Then we describe the functional abstractions of the complete hierarchy bottom-up.

3.1 X/Open FR Authoring

We constructed a functional representation for X/Open Poll-Decide (PD). This experiment explored some of the intentions and issues discussed in Section 2.5. The functional description was constructed using the activities in Table 1.1. Here is a brief account of the process and some authoring decisions:

Understanding the Architecture The architecture was understood from the descriptions in [11], reasonable inference, and background reading about distributed transaction processing protocols. This is the understanding presented in Section 2.4.

Intentions Example intentions were given in Table 2.1. Design intentions were chosen from the understanding to explore representation of general, multi-level explanation involving the problems presented by architecture understanding beyond simple program understanding, e.g. distribution, concurrency, weak execution model, and components. These choices emphasize understanding the algorithmic implementation of the Poll-Decide procedure. Rapide constraints were not used as intentions because they seemed unnecessary and low-level compared to the intentions from understanding.

Languages and Vocabulary At the bottom level, X/Open events and Rapide semantics suggested language, formalism, and vocabulary. These were blended into terminology from successively more abstract domains, terminating with the top-level vocabulary used to discuss Two-Phase Commit in the distributed processing protocol domain. It was possible to completely describe the architecture at each abstraction level with a state machine formalism. The state machines are finite and contain states corresponding to complete states of the architecture at each abstraction level.

Specifications for States The bottom-level states are sets of Rapide events. As appropriate under functional abstraction, lower-level states and causal processes are replaced with abstract states. Abstract states are denoted with evocative names. They are given formal or informal semantics consistent with by the functional abstractions which justify them.

Causal Sequences The bottom-level causal process description captures all possible Rapide behavior. It is a finite state machine which incorporates understanding of the architecture's distributed, concurrent execution. Causal process descriptions at higher levels are generally simplifications of this state machine produced by functional abstraction and abstract states.

Functional Hierarchy The seven level functional hierarchy is described in the following section. For the most part, each level results from a single kind of functional abstraction from the level below. This design is influenced by being an exploratory example, and by desire to make the process and FR simple and pedagogically useful. In other words, we only wanted to change one thing at each step to avoid complexity and to clearly display what was happening. As a result, the functional hierarchy corresponds more to particular functional abstractions than to distinct domains. Particularly, multiple domains are present at each level, and vocabulary and material from lower domains is gradually supplanted by more abstract material.

Abstract Devices The use of abstract devices to modularize the explanation is orthogonal to the preceding issues and activities. The example was simple enough that abstract devices were not needed to control complexity. Furthermore, we they found that heavy use of them would confound the exploratory and pedagogical clarity of the example. In the FR presented, abstract devices are introduced only at the first level of the hierarchy. We discuss further and alternative modularizations and uses of abstract devices in Chapter 5 below.

Validation We will discuss validation when we describe use and application of the FR in Chapter 5.

3.2 The Functional Hierarchy

The Poll-Decide functional representation explains how Two-Phase Commit Poll-Decide is implemented in the Rapide X/Open architecture. Figure 3.1 gives an overview of the FR. The FR can be visualized as seven layers on top of

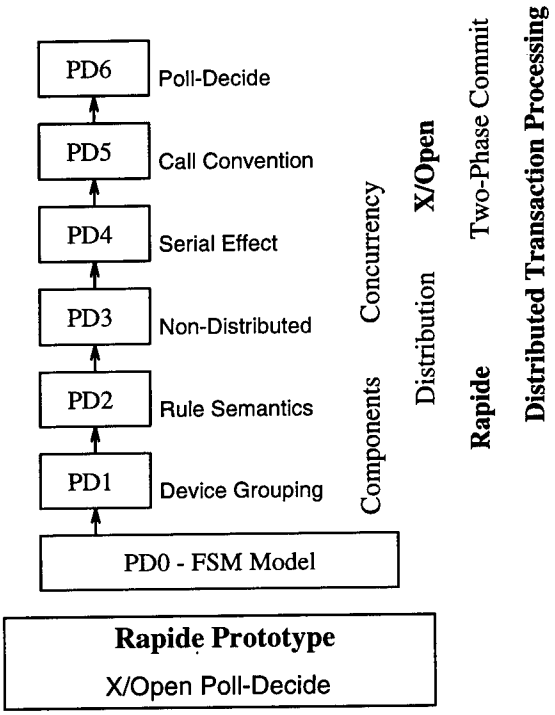


Figure 3.1: Poll-Decide FR Hierarchy

the Rapide prototype architecture. The bottom level, PD0, is a state machine causal process description capturing the architecture's complete behavior. Successively higher levels are successively simpler state machine CPD's. Each one is formed by a particular kind of functional abstraction, shown to the right of each box. The top level, PD6, is a specification of the Poll-Decide part of the Two-Phase Commit protocol. The functional abstractions can each be seen as introducing a more abstract domain of discourse.

The right side of the figure roughly shows some of the domains reflected in the CPD's. Going bottom-up, the FR successively uses abstractions and terminology involving architecture components, Rapide semantics, architecture distribution, architecture concurrency, the X/Open standard and Two-Phase Commit. As observed above, the hierarchy is based on single abstractions rather than

domains, so material from lower domains is gradually supplanted with material from more abstract domains. Furthermore, domains themselves are fuzzy and overlapping. For example, distributed transaction processing intersects some of the domains mentioned previously.

In the remainder of this section we give an overview of the FR by summarizing each level, going bottom-up. We will emphasize the CPD's and the functional abstractions which produce them, without dwelling on the functions. The following sections discuss the representation in detail at each level, including the functions.

PD0 – Architecture Behavior State Machine The bottom level, PD0, completely captures the architecture's single transaction behavior with a model which is a finite state machine (fsm). This model is based on understanding the semantics of the Rapide rule execution model, and on understanding the Poll-Decide algorithm and its implementation in the Rapide abstract behavior and connection rules. Fsm states were constructed from sets of architecture events which occur together. Sequential and concurrent Rapide threads were distinguished. They are modeled with sequential and concurrent sections in the fsm, and corresponding fsm semantics. The PD0 fsm is execution equivalent to the architecture in the sense that an architecture execution and an fsm execution have the same states and state transitions (concurrency non-determinism aside).

Note that PD0, like the architecture, specifies all possible executions or behaviors. Therefore it is a stronger behavior description than a single transaction poset trace, which records a particular execution with particular input. An given execution of PD0 corresponds to a poset trace which would be produced by the architecture.

The PD0 finite state machine is recorded with FR causal process description syntax. States are connected by transitions. The transitions' justifications are given by ByRapide annotations which point to Rapide rules.

PD1 – Abstract Sub-Devices PD0 can be divided into sub-devices corresponding to the architecture components – TM, RM1, and RM2. However, this produces disjoint regions for the polling and decision behaviors in TM. To distinguish these computations, PD1 is created with separate abstract devices or sub-devices for the polling and decision phases. The result is four connected causal process descriptions.

PD2 – Rule Logical States In Rapide a rule precondition can be satisfied by various sets of events. There is no event for acceptance by the interpreter. There is corresponding behavior in the PD0 and PD1 CPD's, where states corresponding to precondition events are immediately followed by states corresponding to postcondition events in the architecture. In PD2 we introduce abstract states corresponding to acceptance in rules B2 and B3. This uses and captures understanding of Rapide rule semantics, and captures anticipation that the new states

will be useful for representing the Poll-Decide algorithm. The result consists of a new CPD with added intermediate states.

PD3 – Distribution Removal Rapide produces events for connection traversal which have the same status as events from other aspects of architecture behavior. For algorithmic understanding, we wish to abstract away these artifacts of architecture component distribution. PD3 eliminates states and transitions due to connections. This uses and captures understanding of distribution in the architecture, and of its representation in connection rules and their events. The result is a simpler CPD for an equivalent non-distributed device.

PD4 – Concurrency Removal Similarly, the Poll-Decide algorithm is obscured by the mechanisms for concurrent polling in the architecture. PD4 abstracts away concurrent polling. This uses and captures understanding of concurrency in the Rapide execution model and in the Poll-Decide implementation The result is a simpler CPD for an equivalent serial device.

PD5 – X/Open Standard Call Removal The architecture was based on the X/Open standard, in which C calling conventions are used to specify the interfaces between components. In the prototype architecture, rules were written to create events for both calls and returns. In a certain sense, these are nonessential artifacts of the standard. To focus on the protocol implementation independent of the calling mechanism, we created a CPD without the states caused by calling and returning. This uses and captures understanding of the roles of calls in the standard and the architecture, and of how they were implemented in architecture rules. The resulting simpler CPD, PD5, gives a view of the algorithm as if it were contained in a single program unit.

PD6 – Poll Decide As a result of these abstractions, PD5 displays conditions leading to alternative logical states and actions. With an understanding of Two-Phase Commit and its realization in the architecture, this can be interpreted and abstracted to concisely give a CPD, PD6, which captures the essence of Two-Phase Commit Poll-Decide. The states and transitions in PD6 are specifications of Two-Phase Commit polling leading to approval and commitment, or to disapproval and rollback. PD6, therefore, uses and captures this top-level understanding, and comprises a specification of Poll-Decide. It is also the top level of an explanation showing how Poll-Decide is implemented in the original Rapide architecture.

We describe this top level and its representation more in the following section. Subsequent sections then discuss the detailed representations of the the bottom level, and of the remaining levels going bottom-up.

3.3 Top Level – PD6 Poll-Decide

Here is a statement of the X/Open Poll-Decide architecture's top-level algorithmic intention:

> *Poll resource managers and decide to commit or rollback the resource operations.*

This understanding is necessary for architecture evolution involving Two-Phase Commit DTP functionality.

This is a succinct statement of the architecture's function because it uses vocabulary with specific technical meaning in the Two-Phase Commit DTP domain. For example, "Poll resource managers" is understood to refer to asking all resource managers for their final approval of transactions which were previously requested by the application program. Furthermore, the whole statement is known to describe the Poll-Decide procedure of Two-Phase Commit, which ensures transaction atomicity. It implicitly contains the definitions, necessary conditions, and proof of atomicity. For example, it is assumed that each resource manager will only give final approval when it can guarantee that the transaction is consistent and atomic with its other transactions and operations.[1]

Figure 3.2 shows PD6, the functional representation of this understanding. PD6 is a causal process description involving the abstract states and functions

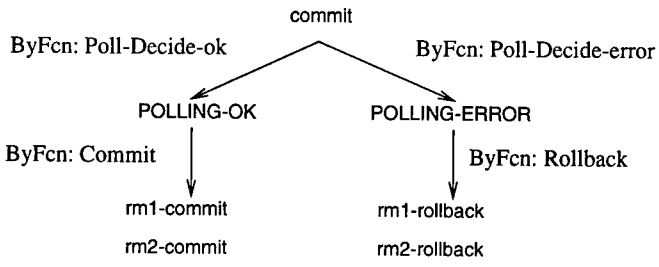


Figure 3.2: PD6 – Poll-Decide Top-Level

at the top of the functional hierarchy of the complete FR (Figure 3.1). It could be paraphrased in English as:

> Following the architecture's `commit` state, the Two-Phase Commit implementation polls the resource managers and decides whether to commit (entering `POLLING-OK` state) or rollback (entering

[1]Note that some of the necessary functionality may not be implemented in the architecture. In this case the FR captures designer intentions which must be realized in the final implementation. The FR's "proof" of atomicity therefore contains assumptions and specifications for the final implementation. For example, here the FR displays the unimplemented intention that the resource managers vote in a way which ensures atomicity. This assumption is a specification which can be verified in the final implementation.

POLLING-ERROR state) the resource operations. Depending on which of these states is reached, the system next enters a state initiating commit or rollback.

PD6 and this paraphrase both connect the general statement of Poll-Decide at the beginning of this section to this specific architecture. PD6 relates the algorithm to the abstract intentions of Poll-Decide and transaction atomicity. Furthermore, the abstract algorithm in PD6 is traceable to its implementation in the architecture.

Contrast PD6 with the Rapide ADL in Figures 2.3 and 2.4. The Rapide ADL description contains no information about the architecture's intentions, is computationally obscure because of the rule-event-poset computation model, and contains many confusing architectural details. The actual Rapide code is far more obscure because of complex syntax, as discussed in Section 2.4. In contrast, PD6:

- States top-level intentions,
- Uses a simple, procedural, local model of computation,
- Gives only essential information.

PD6 is a simple and useful view in the DTP domain. Furthermore, it points to a rich explanation of how the architecture implements Poll-Decide, including the reasoning leading to the top-level specification. This reasoning and implementation is explained in the complete hierarchical FR leading from PD6 to the Rapide code. Since the FR is added on top of the Rapide code, it adds information and value without losing anything which is initially present.

The connection to lower levels is given by the functions in PD6. The functions are not included in the paraphrase above. States and realized transitions are sufficient for a top-level description. More detail can be obtained by referring to the functions which cause the state transitions. For example, the transition from the architecture's abstract `commit` state to the abstract state `POLLING-OK` is caused by the Poll-Decide-ok function. This is shown by the ByFcn annotation in the CPD. The definition of Poll-Decide-ok then shows how committing the transaction is justified and realized in terms of more concrete views and Rapide architectural implementation.

Tracing all the functions downwards through the hierarchy give a complete explanation, presenting all of our understanding of the architecture. This explanation contains captured views, abstractions, and intentions, and suggests simple inferences such as:

1. Concurrent polling computations merge.
2. The `prepare_call` and `prepare_return` states are paired, and could correspond to procedure calling in the X/Open specification domain.
3. The `prepare_call` states produce a Two-Phase Commit "polling" computation.

4. Polling's sole effect is a decision branch in the Two-Phase Commit protocol.

The nature of the explanation will become more concrete as we present the FR hierarchy bottom-up in the following two sections.

3.4 Bottom Level – PD0 State Machine

The bottom level of every functional representation gives the device's primitive, uninterpreted structure and/or behavior. This bottom level must be in FR syntax and conform to FR theory and ontology for describing devices. FR bottom levels typically describe behavior with a CPD.

The architecture we wish to represent is given by Rapide code like that in Figure 2.4. The structure creating behavior consists of the connection and abstract behavior rules extracted in Figure 2.3. These rules describe behavior with semantics given by the Rapide interpreter. Like other kinds of programs, they describe a range of potential behaviors. Actual behavior depends on particular run-time input. An example of run-time behavior is given by the poset trace in Figure 2.5, which is another kind of behavior description.

How can we capture Rapide X/Open behavior in a CPD? There are various possible CPD's which can be chosen according to goals. Writing each possible CPD requires determining states and transitions. For example, the poset can be seen as a CPD where the events are CPD states and the transitions are satisfied dependencies. However, this is a weak description because states are partial, because it covers only a single execution, and because it covers only a single transaction. An alternative CPD, which could also be constructed automatically from a Rapide representation, consists of disjoint segments where rule preconditions and postconditions are states connected by a transition corresponding to a rule firing. This essentially reformats the rules as a CPD, and makes the Rapide execution model implicit in the transition justifications. It has the advantage of generality, being equivalent to the Rapide code for all executions, including multiple transactions. It has the disadvantages of partial states and obscurity due to the disconnectedness of the rules and the implicit computation model.

For algorithmic understanding, we chose another representation that displays important behavioral relationships using total states, for all possible executions of a single transaction. This is stronger that the alternatives in the sense that a CPD total state completely describes the state of the architecture, capturing and facilitating important understanding. It more general than the poset in the sense of capturing all possible executions for a single transaction. It is less general than the Rapide code of its CPD equivalent because it covers only a single transaction. This restriction is necessary to have a fixed number of complete states.

More positively, the single transaction assumption captures an important, essential way of viewing and understanding architecture and DTP behavior. For algorithmic understanding, DTP systems and protocols are typically discussed

with respect to single transactions. This is the case in the X/Open and Rapide literature, much of the DTP literature, and in our descriptions of natural understanding of X/Open earlier in this report. Furthermore, the simulation example in [11], given in Section 2.4, uses a single transaction for analysis.

Figure 3.3 presents PD0, our CPD capturing this Poll-Decide behavior of the Rapide X/Open architecture. PD0 is a state transition diagram for a finite state

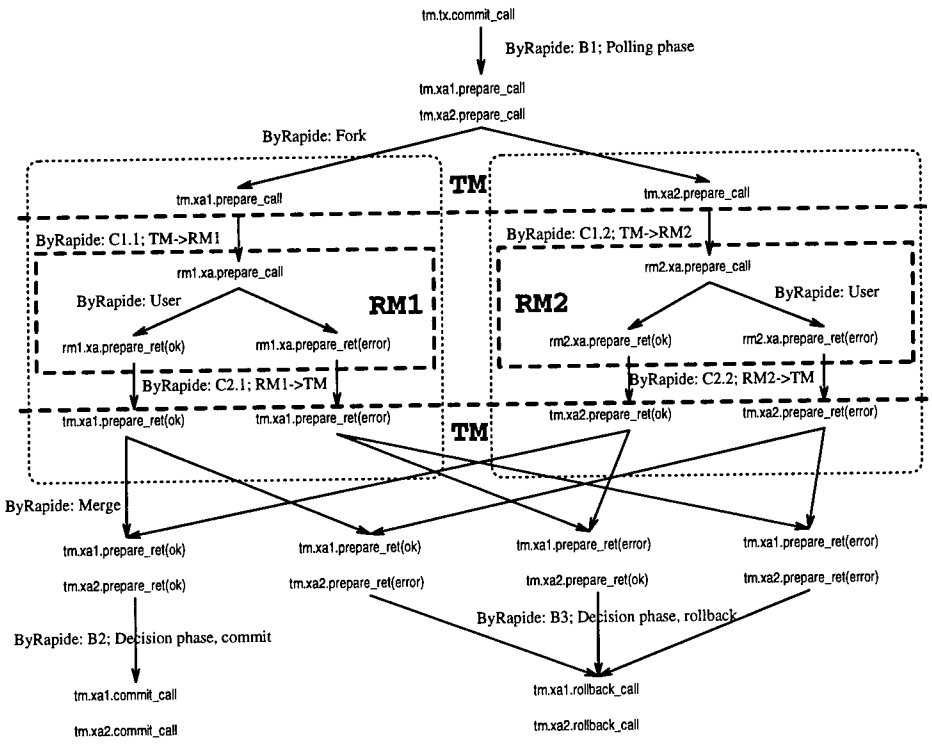


Figure 3.3: PD0 – Poll-Decide Finite State Machine

machine showing all possible executions and internal states involving a given transaction. It is a complete bottom-level description of how the prototype and the architecture can behave.

The *states* of PD0 are the complete, mutually exclusive states that can occur during prototype execution, subject to concurrency. Each state is a set of architecture events. In the figure, for clarity we show only new events, when in fact the CPD states consist of all architecture events that have occurred.[2] For example, the state `[tm.xa1.prepare_ret(ok) tm.xa2.prepare_ret(error)]` means that those two events have occurred. Other events which have occurred,

[2]Events can be removed if they are no longer significant. For example, an event can only trigger a rule in a process once, so it can be removed after all possible triggerings.

such as `tm.tx.commit_call`, may also be in the state but are not shown in the figure.

The *links* of the CPD show the transitions in the fsm. CPD links have *annotations* justifying the transitions. In PD0 all of the transitions are due to connection or abstract behavior specification rules, or to concurrency in the architecture, as described below.

PD0 is divided according to the architecture components. The states above the top heavy dashed line and below the bottom heavy dashed line involve the transaction manager, TM. The states in the heavy dashed boxes involve the resource managers, RM1 and RM2.

Now we will further explain PD0 and its treatment of concurrency by narrating its transitions for the architecture execution described in Section 2.4 and recorded in the poset in Figure 2.5. PD0 covers the architecture beginning with its `tm.tx.commit_call` event. PD0 begins in its `[tm.tx.commit_call]` state. PD0 can then change to the `[tm.xa1.prepare_call tm.xa2.prepare_call]` state. This state corresponds to the architecture after those two new events have occurred. The link in PD0 has the annotation: `ByRapide: B1; Polling phase`. This represents the transition being caused by Rapide rule B1. "Polling phase" is a comment from the rule in the architecture which is added as a comment in the annotation.

The lightly dashed boxes in Figure 3.3 are sub-fsm's that execute concurrently. From state `[tm.xa1.prepare_call tm.xa2.prepare_call]`, PD0 enters state `[tm.xa1.prepare_call]` in the left sub-fsm and state `[tm.xa2.prepare_call]` in the right sub-fsm. This represents forking into concurrent execution in the architecture, shown by branching arrows with the annotation `ByRapide: Fork`.

The left fsm captures an execution thread involving resource manager RM1. Within the left sub-fsm there are transitions corresponding to a call passing across the connection from TM to RM1, user interaction requesting an "ok" or "error" return, and either return passing across the connection from RM1 to TM. Each is annotated with a Rapide rule or user interaction as a cause. The right fsm is similar, capturing a concurrent execution thread involving resource manager RM2. Each sub-fsm can finish in one of two states.

Following completion of concurrent execution in both sub-fsm's, PD0 enters a state corresponding to one of the possible pairs of finishing states of the sub-fsm's, e.g. PD0 state `[tm.xa1.prepare_ret(ok) tm.xa2.prepare_ret(error)]`. The states signify that both events have occurred in the architecture. These transitions correspond to concurrent execution ending and becoming a single thread in the architecture. Each of the possible PD0 states is at the heads of two joining arrows, one from each sub-fsm, with annotations `ByRapide:Merge`.

Finally, the state with both "ok" returns has a transition to a state corresponding to commit_call events for both RM's, with an annotation giving the cause as Rapide rule B2. The other three states after merging each have at least one "error" return. They are all followed by transitions to the state corresponding to rollback_call events for both RM's, with an annotation giving the cause

as Rapide rule B3.

In summary, PD0 is an FR causal process description for the behavior of the X/Open architecture. It is behaviorally equivalent to the architecture for all possible executions of a single transaction, in the sense that its states and transitions completely capture the possible states (sets of events) and transitions in the architecture. The causal process description is a finite state machine. PD0 has many conceptual and representational advantages over the original Rapide code. In the following section, we will see how it provides the basis for functional abstraction as we present the remainder of the FR bottom up.

3.5 Intermediate Levels

The rest of the FR is a hierarchy of CPD's. The CPD at each level is a finite state machine similar to PD0. However, higher level machines are successively simpler, and introduce abstract states that may not be sets of architecture events as in PD0. Furthermore, functions in higher level machines may point to lower machines and/or other justifications rather than Rapide rules and interactions.

3.5.1 PD1 - Abstract Sub-Devices

PD0 can be divided into sub-devices corresponding to the architecture components – TM, RM1, and RM2. These components are based on distribution in the architecture and the X/Open standard. Other components may be better for creating and representing understanding. FR allows the creation of such new components, called *abstract sub-devices*.

In the PD0 state machine above we see there are disjoint regions for the polling and decision behaviors in TM. To distinguish these computations, PD1 is created by dividing PD0 with separate abstract sub-devices for the polling and decision phases, as shown in Figure 3.4. Syntactically, PD1 is a causal process description consisting of four sub-CPD's connected at common states (boldface). Each sub-CPD is given a name expressing an understanding of its function appropriate for this level of the hierarchy.

PD1 introduces a functional decomposition in place of the original architectural decomposition. This decomposition is suggested by the causal process description representation of PD0. It is not apparent in the architecture code (although it is suggested by comments). This functional decomposition will be useful in forming and representing functional understanding in the higher levels of the FR described below.

3.5.2 PD2 - Rule Logical States

Functional understanding involves "reading between the lines" to see intentions that are not explicit in the architecture. This includes reading between the explicit states of the architecture to see logical states corresponding to intentions.

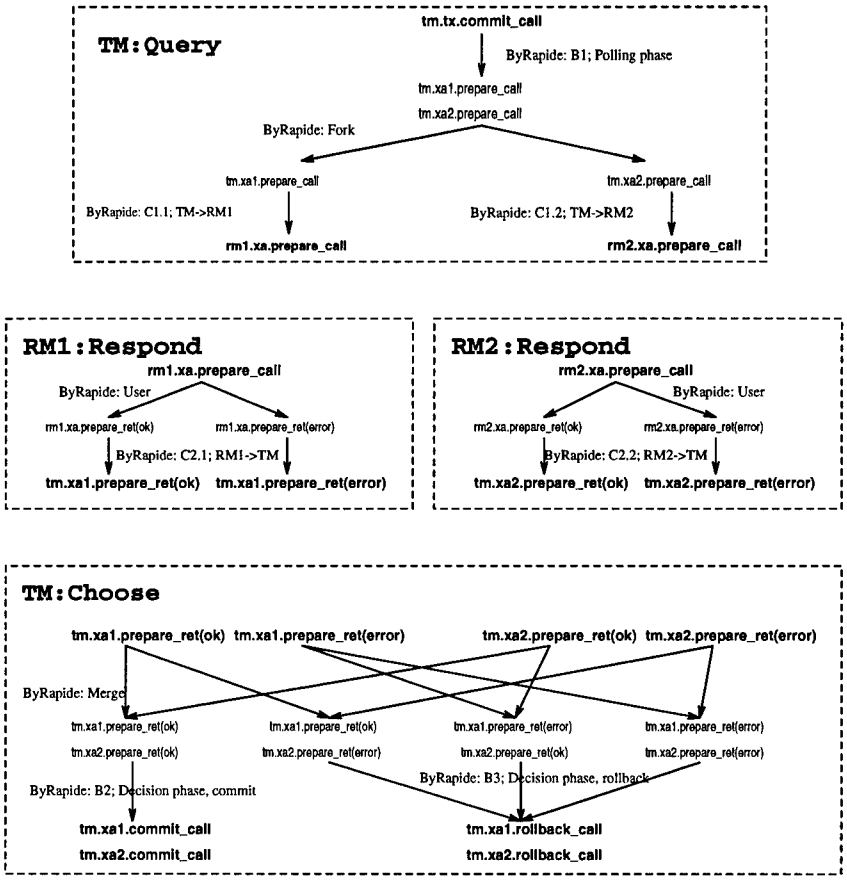


Figure 3.4: PD1 - Functional Abstract Sub-Devices

PD0 makes two such states explicit in a CPD that represents understanding of intentions that are implemented using Rapide rule semantics.

In PD0 and TM:Choose in PD1 we see transitions to the final states caused by rules B2 and B3. Semantics of Rapide rules and their interpretation are implicit in these transitions. The interpreter first checks rule preconditions, leading to an "accept" or "don't-accept" state in the interpreter. When "accept" occurs, postcondition events are generated. Furthermore, triggering depends on satisfying the rule's precondition pattern, which is given in the fairly elaborate Rapide pattern language. For example, any of the three states in PD0 and TM:Choose which contain at least one "error" return will trigger B3. Reading between the lines here means understanding that acceptance captures a useful logical property, "some_error", shared by three different states.

Such understanding is represented in the TM:Choose sub-device of PD2, shown in Figure 3.5. Boldface shows the changes from PD0 and PD1. The

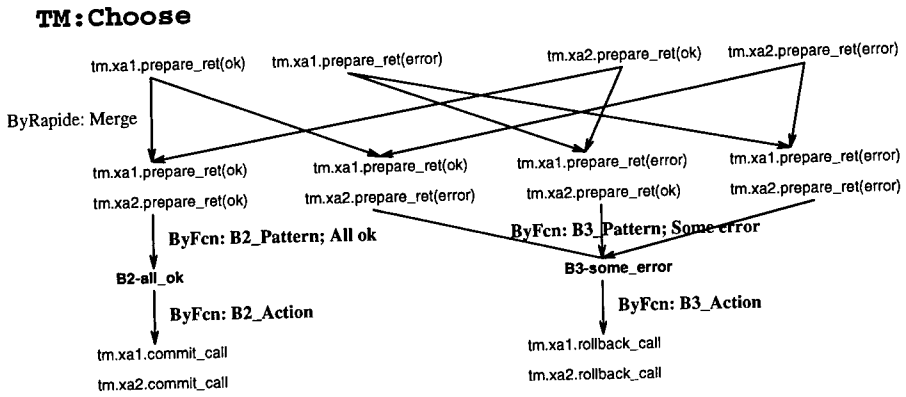


Figure 3.5: PD2 - Rule Semantics Abstractions

additional states `B2-all_ok` and `B3-some_error` have been added as abstract intermediate states. Therefore implicit states and intentions are represented explicitly. PD2 is equivalent to the architecture and PD0 in the sense that it reproduces the original behavior (but also added new states and transitions).

This requires understanding Rapide rule pattern semantics. For example, the precondition of rule B3, `(?x in xid) (?i in 1..NumRMs) XAs(?i).prepare_ret(?x, error)`, must be understood as matching sets of `tm.xa<i>.prepare_ret(?x, error)` events, where some event has an error return.[3] This understanding is represented by the functions `B3_Pattern` and `B2_Pattern` which annotate the transitions to the new states. The functions are followed by appropriate comments.

[3]This is the original rule with a transaction identifier parameter x of type xid. Recall that the transaction identifier is implicit on our usual event notation for brevity.

Similarly, links are added connecting the new abstract states to the original states which are consequents of the rules. These links are annotated with functions `B2_Action` and `B3_Action` to represent that the new logical states cause the original events, again capturing understanding of Rapide rule semantics.

We will see the usefulness of the new states higher in the FR. For example, the "some_error" logical property will be given Two-Phase Commit domain interpretation as the intention of recognizing when some resource manager disapproves finalizing the transaction.

3.5.3 PD3 - Distribution Removal

The X/Open standard and Rapide architecture emphasize that TM, RM1, and RM2 are geographically distributed components. Connections between them are defined in the architecture by connection rules. Rapide produces events for connection traversal which have the same status as events from other aspects of architecture behavior. Distribution and intermingled connection and behavior events complicate architecture understanding. For algorithmic understanding, we wish to abstract away distribution and its connection machinery.

PD3 is a CPD for a non-distributed version of the Rapide X/Open architecture, shown in Figure 3.6. In PD3 all of the connection machinery and its

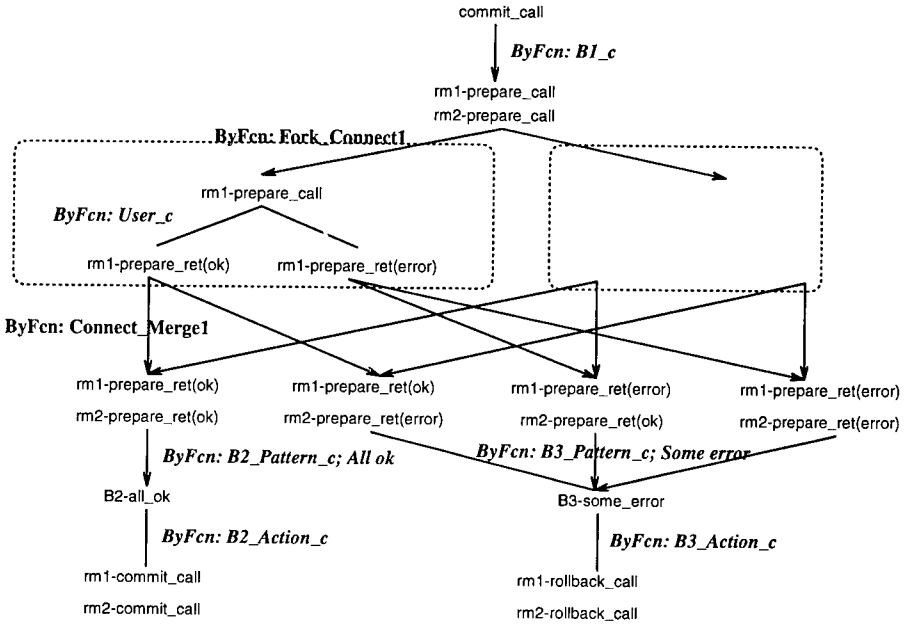


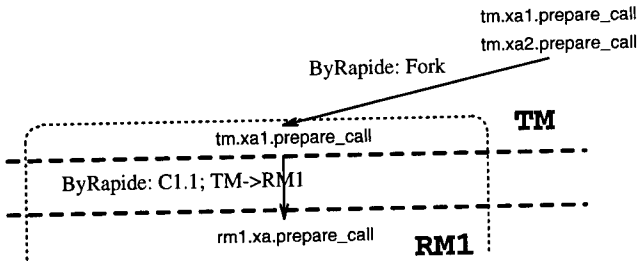
Figure 3.6: PD3 - Connection Elimination

consequences have been removed, producing a simpler representation. States

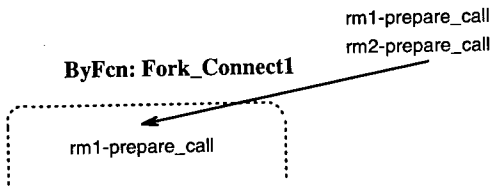
and transitions due to connections have been eliminated. Most remaining states have new names, where connection terminology has been removed from event names. There are also new functions throughout.

The PD3 fsm is functionally equivalent to PD2 and the architecture in the sense the same function is being computed. Particularly, the final states of PD3 correspond to the final states of PD2 and the architecture for all input. It is not strictly execution equivalent because it does not pass through the same states and transitions. Actually, PD3 passes through states and transitions that correspond to a subset of the states and transitions in PD2 and the architecture, and through added abstract states.

This simplification captures understanding of distribution in the architecture, and of its implementation with connection rules. This functional abstraction is represented by the new functions. The most important new functions, shown in regular boldface, show how connections are eliminated. For example, in PD0, PD1, and PD2 we have:



Function `Fork_Connect1` represents understanding of the connection in the architecture, and of how it may be eliminated. The transitions and states involving the connection above are replaced in PD3 with:



Note that that states based or original Rapide events are replaced with states with new names which eliminate connection terminology, e.g. the "xa" service designation. While indications of distribution are eliminated, "rm1" is still present to distinguish the two RM's in the non-distributed view. Note also that the finite state machine still has concurrent sections, as indicated by the dotted box. Other similar functions are shown in regular boldface.

Because all state names were changed to non-distributed conventions, all states now have new names, and are abstract states. This means all functions must be changed to explain the new names. This is included in the functions above. The remaining functions do not directly eliminate connections, but are changed for new names, and are shown in italicized boldface in Figure 3.6.

3.5.4 PD4 - Concurrency Removal

After distribution is removed, PD3 still contains concurrent processes that obscure the essential Two-Phase Commit Poll-Decide algorithm. In the architecture, the transaction manager polls both resource managers concurrently. This is not an essential aspect of Poll-Decide.[4] Concurrent polling complicates representation and understanding.

PD4 abstracts away concurrent polling, as shown in Figure 3.7. Compare

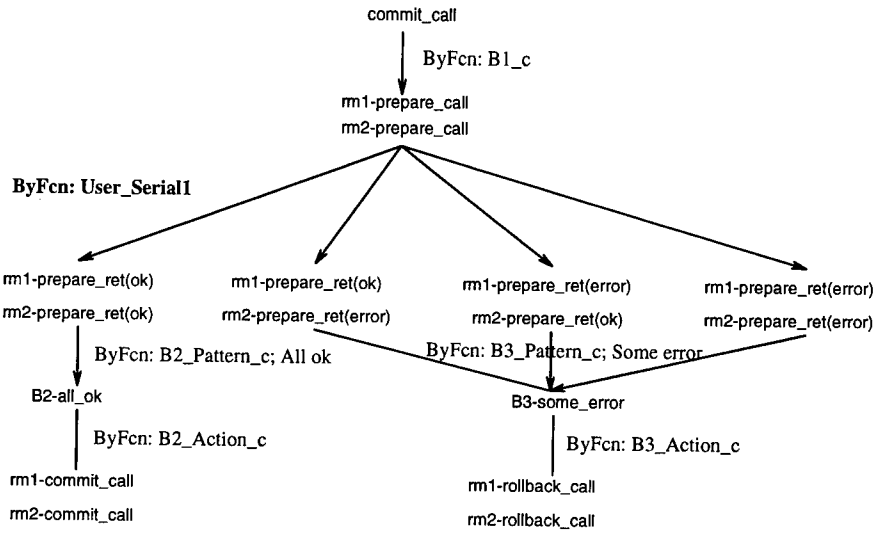


Figure 3.7: PD4 - Serial Effect From RM's

PD4 with PD3 in Figure 3.6. PD4 eliminates the concurrent regions and the states and transitions involving concurrent polling. The result is a CPD which is a conventional finite state machine, with no concurrent sub-fsm's. PD4 is functionally equivalent to PD3. It is execution equivalent to the subset of PD3 which is preserved.

PD4 uses and represents understanding of concurrency in the architecture and the algorithm. This is captured by four functions like `User-Serial1` which replace the concurrent regions. These functions' justifications contain understanding of concurrency in the Rapide execution model, and of its role in this specific calculation. For example, function `User-Serial1` causes a transition to

[4]Of course Two-Phase Commit and Poll-Decide are motivated by concurrent resource operations, involving both multiple transactions and multiple resource operations for the same transaction. However, in the single transaction view, Poll-Decide takes place after these concurrent operations are finished. Polling doesn't have to be concurrent. The architecture presumably used concurrent polling either because of 1) an architectural decision to use concurrent polling for efficiency, or 2) to avoid specifying irrelevant detail, because Rapide rule code is concurrent by default, requiring extra coding effort to make it serial.

state [rm1-prepare_ret(ok) rm2-prepare_ret(ok)] when the result of concurrent polling is approval from both resource managers. This is justified by understandings such as: 1) polling forks from and merges into a single sequential thread, 2) the time orderings of the concurrent processes don't affect results, i.e. they are serializable, and 3) the user interactions in both RM's determine the state after concurrent polling.

Once again, functional abstraction created a representation that is simpler and closer to to top-level intentions regarding Poll-Decide. Implementation details which are irrelevant at this abstraction level were removed using knowledge and understanding of the architecture, Rapide, and the concurrency domain. The understanding which permitted abstraction is recorded in the function justifications. They capture subsidiary intentions, which have implementations that can be traced down the hierarchy.

3.5.5 PD5 - X/Open Standard Call Removal

For the purpose of understanding Poll-Decide, concurrent polling is an unnecessary and distracting artifact of how Poll-Decide was implemented in the architecture. We removed it and recorded understanding of its role and implementation. PD4 contains one remaining artifact of the architecture – C calling conventions.

Because the X/Open standard described interfaces using C calls and returns, Kenney used them to define the interfaces between operations of the architecture. This causes an call event to be generated at the beginning of an operation, and a corresponding return event to be generated at its conclusion. For example, in PD4 states contain paired <rm>prepare_call and <rm>prepare_ret<code> events from the polling call to and returns from resource managers RM1 and RM2. Poll-Decide can be implemented without the equivalent of subroutine calls and returns, so the calling machinery can be seen as an unnecessary and distracting artifact.

As in the preceding sections, we use functional abstraction to simplify and clarify the representation. PD5 abstracts away calls and returns, as shown in Figure 3.8. Treatment is similar to the functional abstraction which produced PD3. PD5, in comparison with PD4 (and lower levels), has eliminated the calling machinery and its consequences. States and connections due to calling have been eliminated. Most remaining states have new names, where calling terminology has been removed from event names. There are also new functions throughout.

3.5.6 PD6 - X/Open Poll-Decide

In our bottom-up narration, the FR hierarchy culminates in the top level, PD6, which is a specification of Poll-Decide in the Two-Phase Commit and distributed transaction processing domains. PD6 is presented and described in Section 3.3 above. It is reproduced below for comparison with PD5.

The representation is organized by functions, each of which relates functionality in higher levels to behavior and other justifications from lower levels and

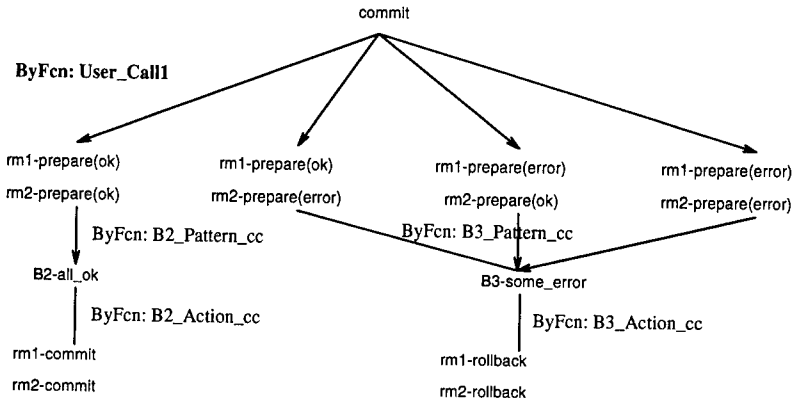


Figure 3.8: PD5 - X/Open Calling Conventions Removed

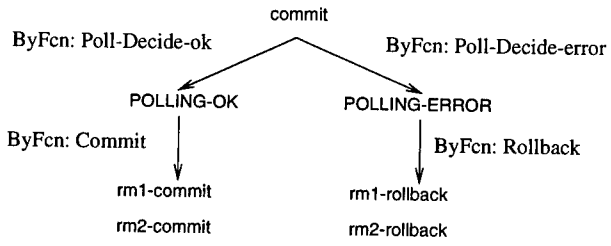


Figure 3.9: PD6 – Poll-Decide Top-Level

domains. Note that functions can be replaced with alternative realizations and justifications without affecting the higher level. This is part of what "functional" and "abstract" mean. Note also that functions are useful for non-function understanding and reasoning. For example, they provide an attachment point for rationales. With the functions creating PD5, for example, the author could record the rationale for concurrent polling.

Now that the complete hierarchy has been presented, we can appreciate the explanation that extends downward from PD6. All of the understandings, subsidiary intentions, implementations etc. discussed in the preceding sections are preserved and accessible in the FR. This material constitutes a rich explanation of the architecture in terms of intentions and domains at various level. The next chapter describes how the explanation can be automatically accessed, following formalized relationships in the FR, in response to particular questions and needs. The semantics of FR can be described using these relationships.

3.6 Summary

This chapter described a functional representation for the X/Open Rapide architecture which captures understanding of Two-Phase Commit Poll-Decide. The FR is a hierarchy produced by various functional abstractions in various domains. Each level is a causal process description that is a finite state machine for complete states of the architecture. The bottom level completely captures architecture behavior for single transactions. Successively higher levels simplify, refine, and focus the description towards the top-level specification of Poll-Decide. Abstraction between levels is given by functions, which identify functionality and intentions in various domains. Function justifications capture the understanding that enable the architecture to be understand bottom-up. The also reproduce the understanding needed to implement the architecture top-down. The complete FR captures many kinds of an understanding in a rich explanation. It can be exploited in various tools and tasks that require such understanding.

Chapter 4

Rapide Explanation Tool

A functional representation captures understanding of a device. This understanding is in the form of an explanation of how abstract functionality is realized in the device. As discussed in Chapter 1, writing an FR for an architecture creates and systematizes understanding for documentation and communication. Beyond this, the FR has specific benefits at two levels:

1. The FR provides explicit answers to certain questions.
2. These answers are necessary or useful for architecture evolution tasks.

This chapter discusses important question classes, and procedures for easily obtaining answers from the FR. These answers are the primitive elements of the FR's complete explanation. Answering such questions is a general quality of FR's and architecture FR's, independent of ultimate application. Chapter 5 describes how answers delivered for architecture questions can be applied to improve architecture evolution.

More specifically, here we describe how the explanation can be automatically accessed in response to particular questions and needs. Answers are delivered based on the relationships in the FR. We first give a generalized model of FR's, identifying the major entities and relationships. We then give a catalog of question types which may be answered and procedures for answering them. A practical explanation tool is described. We discuss its use for navigating the FR and providing needed answers and explanations on demand.

4.1 Delivering Explanations From FR's

Explanations are assertions and inference links which connect intentions to each other, and which connect intentions to the architecture. There are many possible explanations for an architecture.[1] An FR for an architecture makes explicit

[1]There are infinite possible explanations for the architecture because there are infinite valid assertions and inferences that can be made about it. More practically, there are many useful views, models, theorems etc. for the architecture.

one of the possible explanations. Given a particular FR, we say that it is a complete explanation in the sense that it contains everything the author wanted to include. The author deemed it sufficient for his or her purposes, and stopped adding new material. We consider this FR to be the entire explanation universe for the purposes of this chapter.

Still, FR's for even simple devices and architectures can be large and complex. For example, the X/Open FR in the preceding chapter has many states, functions, and components in seven layers. In textual functional representation syntax, the FR is voluminous and the relationships are obscure. Therefore, in the previous chapter we used drawings for individual CPD's. To be clear, we could only display a single, greatly simplified CPD at a time. It is not practical to present the complete FR graphically on normal-sized paper. Size aside, the relationships, including function definitions and CPD transitions, can include multiply diverging and converging branching. For these and other reasons, the complete FR and explanation cannot be presented or comprehended at one time.

Rather, the FR user must focus on parts of the FR, as dictated by his or her current needs. Parts of the complete FR or explanation are themselves explanations. At a certain size and complexity they become comprehensible and useful nuggets of assertions and inferences. The smallest explanations are the primitive relationships of FR. The FR primitive relationships can be seen as answering important questions. In this chapter we will describe delivering explanations in terms of these primitives.

Figure 4.1 summarizes this. A large and complex complete FR or explana-

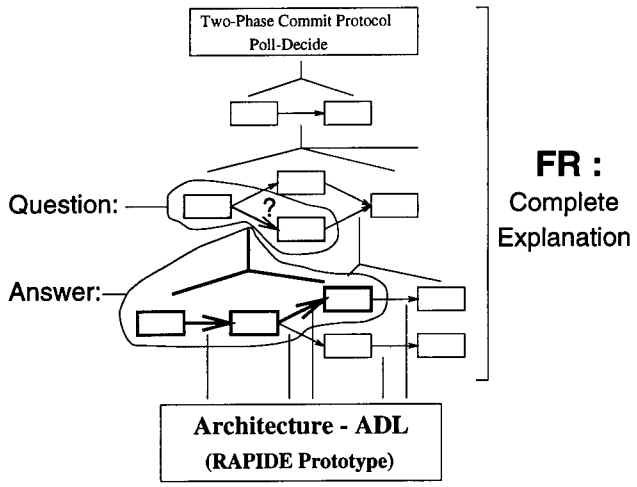


Figure 4.1: Delivering a Primitive Explanation From an FR

tion is shown with a simplified depiction. Delivering a primitive explanation involving the indicated state transition requires answering the question, "How

does the transition occur?" The state change is achieved by a particular abstract function. The function's justification describes how the transition is achieved by lower level behavior, principles etc. Therefore the question is answered by presenting the function definition and its connections in a lower CPD. The function and function justification are "How" and "How Implemented" relationships between the questioned transition and its realization.

In general, explanations can be delivered by extracting primitive relationships in the FR based on their semantics. This provides answers to simple questions. More complex questions may cause larger explanations to be constructed and delivered by combining primitive answers. In this chapter we will mostly describe the primitive relationships of FR and the questions they answer.

4.2 FR Entities and Relationships

The FR language is described in [2], [7] and elsewhere. We will give a simplified FR-Rapide description based on [7] and FR-Rapide extensions.

To illustrate this description, we will use the generalized example functional representation in Figure 4.2. The figure includes the elements of the X/Open

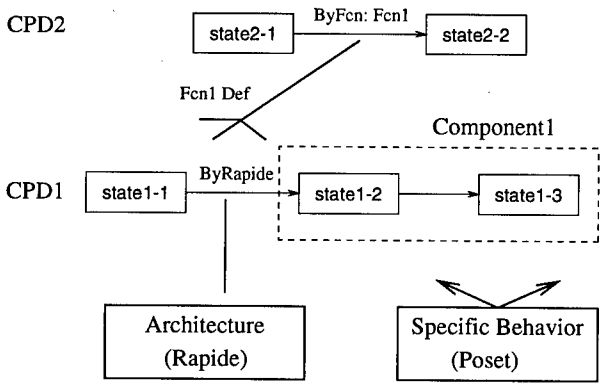


Figure 4.2: Generalized Functional Representation

FR in the preceding chapter.

The primitive boxes are **states**, with state names or predicates like "state2-1". States are connected with directed links or **transitions**. Sets of connected states and transitions form state transition diagrams called causal process descriptions or **CPD's**.

There are various grouping constructs involving CPD's. A **named CPD**, e.g. "CPD2", may comprise a complete **level** in the FR's explanation hierarchy. CPD's contain named sub-CPD's called **components**, e.g. "Component1". Logically, components, levels, and larger aggregations in the FR, and the complete FR are all **devices**.

Transitions have various annotations. The most important annotation is **ByFcn**, which gives a function name with a corresponding **function justification** or definition, e.g. "Fcn1" and "Fcn1 Def". Function definitions have various forms which we won't describe here. Typical elements in the definitions are preconditions and postconditions (If and ToMake), and definition elements describing a realization of the function in terms of CPD's, components, functions, definitions, inferences, domain principles etc.

Other annotations are specific to Rapide and architectures. **ByRapide** connects a transition to a Rapide rule by which it is realized. **InPoset** connects states, transitions, and CPD's to parts of Rapide posets which record instances of those FR elements in a particular Rapide execution.

4.3 Questions and Answers

The FR syntactic description above included relationships that are specified explicitly in FR's, e.g. transition(<state>, <state>). There are also implicit relationships which can be easily inferred from the syntactically explicit relationships.

Questions are partly specified relationships. *Answers* are bindings to the missing elements, which are provided by the FR. For example, the question:

```
Q> What state makes a transition to state2-2?
```

is a natural language form of the partly specified relationship: transition(?, state2-2). It is answered:

```
A: state2-1 makes a transition to state2-2.
```

by finding in the FR that "state2-1" has a causal link to state2-2 and can be bound to "?". Similarly, the question:

```
Q> What is the relationship between state2-1 and state2-2?
```

is a natural language form of ?(state2-1, state2-2). It is answered by binding "transition" to "?".

There are many syntactic relationships in FR's. Such questions correspond to the explicit and implicit relationships in the FR, along with possible missing elements. Therefore there are many such questions that can be answered by the FR. It would be possible to construct a general facility to answer meaningful questions, e.g. like a relational database query language.

However, there is a relatively small set of relationships and questions that are especially important for architecture evolution tasks. Some important question classes are shown in Table 4.1.

First we will consider questions involving standard FR elements. Then we will discuss questions involving extensions for FR-Rapide. In both cases, simple answering procedures will be given for each question type.

1. Where is state1-2?
2. When does state1-3 occur?
3. What does Fcn1 do?
4. How is state2-2 achieved (starting with state2-1)?
5. How is Fcn1 implemented?
6. Why is state-1 used?
7. Why is Rapide rule B1 used?
8. Where is poset event explained?

Table 4.1: Important FR and FR-Rapide Questions and Relationships

4.3.1 FR Question Classes

FR question classes involve FR elements that are used in FR's for all kinds of devices, not just architectures. Therefore the following discussion applies to architectures, and to other kinds of devices as well. An explanation tool delivering answers to these questions can be used for both architectures and other kinds of devices represented by FR's.

Where Questions – Syntactic Context

Where questions refer to objects' locations and contexts in the FR. An entity may be located in various, possibly nested contexts.

A simple Where question is answered by finding the object and giving an immediately enclosing context:

```
Q> Where is state1-2?

A: state1-2 is in component Component1.
```

Nested contexts can be retrieved by subsequent questions:

```
Q> Where is Component1?

A: Component1 is in CPD CPD1.

Q> Where is CPD1?

A: CPD1 is the bottom level of the FR.
```

More complex question forms and answering procedures can be used to retrieve nested and/or multiple contexts with a single question.

Where explanations are based on Is-In relationships in the FR. There are corresponding questions based of the inverse relationship, Contains.

When Questions – Causation

When questions retrieve immediate causal relationships. Recall that FR and Rapide are based on causation rather than time. Therefore, asking "When does <state> occur?" is asking about what causes the state. "When" refers to sufficient cause rather than a particular moment in time. It entails logical and temporal order. Furthermore, FR considers states to be the causes of subsequent states.

A simple When question is answered by finding the questioned state and returning state or states that have a transition to it:

```
Q>When does state1-3 occur?

A: state1-3 depends on state1-2.
```

More complex question forms and answering procedures can be used to retrieve causal sequences and branching within a CPD.

When explanations are based on the May-Cause relationships in the FR. There are corresponding questions based on the inverse relationship, May-Depend-On.

What Questions – Function Effect

What questions retrieve functionality represented in the FR. Simple functionality is a state change from a function's precondition state to its postcondition state. The states describe the net effect of the function – what it does.

A simple What question is answered by giving the precondition and postcondition states of a function:

```
Q> What does Fcn1 do?

A: Given state state2-1, function Fcn1 achieves state state2-2.
```

Note the difference between What questions and When questions. When questions refer to arbitrary state transitions, which may be of various types. In contrast, What questions refer only to transitions caused by functions, where transition links have the ByFcn annotation.

What explanations are based on the Function-Achieves relationship between a function and its state change effect. The inverse Achieved-By-Function relationship is important enough to distinguish in the following question type.

How Questions – Function Name

How questions retrieve the names of functions which realize state change functionality. They are an inverse of What questions.

The simplest How question gives the name of a function which achieves a given state change effect:

```
Q> How is state2-2 achieved (starting with state2-1)?

A: Given state state2-1, function Fcn1 achieves state state2-2.
```

Th function name is obtained from the ByFcn annotation on the transition. For many purposes, the function name may be evocative enough to be a useful explanation.

How Implemented Questions – Function Realization

How Implemented questions retrieve descriptions of functions' realizations:

```
Q> How is Fcn1 implemented?
```

The question is answered by giving the function's definition or justification.

The form of the answer depends on the form of the function justification and the detail sought. Recall that function justifications can use CPD's, components, functions, definitions, inferences, domain principles etc.

The simplest answer can be given when a function is implemented by a named CPD:

```
A: Fcn1 is implemented by CPD CPD1.
```

More general and detailed answers can be given by describing the implementing CPD in various ways. For example:

```
A: Fcn1 is implemented by a CPD with the causal sequence:
    state1-2 -> state1-2  -> state1-3
```

Giving all the states is general, but may be confusing when there are many states and/or the states are obscure. Alternatives depend on the particular function definition. For example, it may be possible to say that the function is implemented by a component.

On the other hand, there is more to the justification than just the states. It may be useful to also convey the annotations, componentization, level, context, domain connections etc.

Depending again on the function definition and its nature, the diagrammatic form of the CPD may be the preferred answer. This requires departing from natural language explanation and the question answering model of explanation. We discuss alternative explanation modalities in Section 4.4 below.

How Implemented explanations are based on the relationship between function names and definitions. The inverse relationship is important enough to distinguish as the following question type.

Why Questions – Abstract Intentions

Why questions recover intentions that are realized by abstract implementations. Elements in a function justification are associated with the whole function and functional abstraction:

```
Q> Why is state-1 used?

A: state-1 is used to implement function Fcn1.
```

The question is answered by giving the function's justification or definition.

The form of the answer depends on the form of the function definition and the detail sought. Recall that function definitions can use CPD's, components, functions, definitions, inferences, domain principles etc.

More detailed answers are possible. For example, more about the query element's role can be shown by giving the function definition. More about the function can be shown by giving its functionality (as with What questions). However, the simplest answer may be preferable first, especially when the function name evokes functionality and other questions can be asked for more details.

Stronger answers may or not be possible. For example, Why questions may be seen as seeking explanation of the way the querying element interacts with other elements in the function definition. Or they may be seen as seeking design rationale. Such interpretations of why questions depend on the function justification, and on the kind of explanation sought.

4.3.2 FR-Rapide Question Classes

The question types above apply all FR's, for all kinds of devices. Fr-Rapide extends FR with bits of Rapide-specific syntax designed to specialize certain explanations to Rapide. They create FR-Rapide specific question classes. Here are some examples:

How Questions – Rapide Implementation

At the bottom level of the FR, transitions are caused by Rapide structure, not abstract functions. We give a distinctive answer form for How questions in this case:

```
Q> How is state1-2 achieved (starting with state1-1)?

A: By Rapide rule B1 [- <rule>]
```

The form of the question is the same as the general How question above. However, at the bottom level states are Rapide events, and transitions are annotated with ByRapide and a pointer to an element of the Rapide code, typically an abstract behavior rule or a connection rule. The presence on a ByRapide annotation instead of a ByFcn annotation triggers the Rapide implementation answer form. The answer form is similar combined answers to a How and a How Implemented question above.

How questions with Rapide implementations are based on the relationship between bottom-level FR transitions and implementing Rapide code. The inverse relationship is important enough to distinguish as the following question type.

Why Questions – Intentions for Rapide Code

Why questions should be answered for Rapide implementations as well as for abstract implementations. Given a Rapide element, intentions which the Rapide code implements are recovered from the FR.

```
Q> Why is Rapide rule B1 used?

A: Rapide rule B1 is used to achieve state1-2 (starting with state1-1)
```

The question is answered by giving parts of the FR that use the Rapide element.

In this case the Rapide element is a rule which implements a single transition. There are more complicated cases. Given Rapide rules may be involved in various transitions in the FR. Rapide elements besides rules can be queried, e.g. "Why is Rapide ¡event¿ used?" The general answering strategy identifies all places in the FR that directly depend on the Rapide element. These relationships are more restricted than the relationships involved in the general Why questions above.

Rapide Behavior Questions – Intention Poset Mappings

Unlike most other FR domains, Rapide provides an explicit, standardized representation of behavior – posets. For this behavior record to be useful, it must be mapped to both Rapide code and abstract intentions, e.g. during analysis or debugging. Presumably the Rapide environment records pointers to causing code in posets, and provides tools for relating traces to code. The user is then left to figure out the code's intentions and relate them to correct or incorrect behavior in the poset.

With FR we have a record of intentions. Therefore it should be possible to give some intentions for given behavior in the poset. This could be valuable for many Rapide activities involving interpreting posets.

Questions involving poset elements could be answered using Rapide methods to map from poset behavior to Rapide code, and then using the Why question above to find parts of the FR which depend on that code element. However, this is circuitous and crude. For example, since there may typically be a many-to-relationship between events and code, the Why question could return FR elements for all the events related to a rule, not just the FR elements related to the single query event.

Therefore we suggest a question type that returns FR elements related to poset behavior elements. For example:

```
Q> Where is <poset event> explained?

A: <poset event> is in state1-1
```

There are various cases, but the answering procedure can be similar to that for Where questions above. In Where questions a given FR element is located,

perhaps anywhere in the FR. In Rapide Behavior questions, the query item is a behavior element, not an FR element. So it is necessary to look within FR elements, and the behavior element may be found in multiple places in the FR. However, the behavior element will only be found low in the FR, in concrete states or transitions. For example, assume a query is made using a poset event. The event may be found as part of one or more FR concrete states.

Rapide Behavior questions relate behavior to concrete parts of the FR containing Rapide events and transitions. Other question types can then be used to expand the explanation with abstract intentions, including functions and abstract states.

There is obviously an inverse form of this question type. Given an intention in the form of a concrete FR element, we can see where it is manifested in a poset behavior description.

Rapide Behavior questions provide the explanation primitives for many Rapide activities involving simulation and poset behavior traces. Tasks like architecture debugging, reverse engineering, analysis, and design verification can benefit from captured understanding that relates behavior to intentions. This is analogous to uses of FR with simulated behavior in physical devices, e.g. as described for simulation and design verification in [10] and [5].

4.4 FR-Rapide-Explain Tool Design

We now consider the design of a practical tool for delivering explanations. Recall that we are emphasizing explanation primitives, as shown by the question types above. Larger explanations can be constructed from these primitives.

We suggest a tool design that combines two interface paradigms:

1. Question Answering

2. Graphical Hypertext Navigation.

4.4.1 Question Answering

Question answering was described above. That discussion was partly intended to describe FR explanation semantics in terms of primitive elements of understanding and explanation. The discussion was also intended to give a model of FR content and explanation delivery.

Still, question answering has attractive qualities for practical tools. For example, the interface itself is easy to implement. Furthermore, interaction and output is self-documenting, so little training is required. This is because we already know meanings of the language involved. Only a small, structured subset of English is used. Therefore it is possible to quickly learn precise FR-Rapide semantics as specialization of initial intuitive semantics.

Some of the advantages for simple novice use become disadvantages for more complex explanations and expert use. For example, typing questions is slow,

and natural sounding answers seem verbose with familiarity. Similarly, question answering has narrow bandwidth and single, textual modality.

4.4.2 Graphical Hypertext Navigation

Graphical hypertext navigation offers complementary qualities. It is an alternative paradigm for viewing an FR. Navigation is like a human reading a large, graphical depiction of the FR hierarchy. We can move freely over the FR, expanding or contracting context as needed. Multiple relationships and complex CPD's are visualized from diagrams. Complexity of the presentation can be controlled, e.g. by clicking to expand of contract the amount of detail presented. All this contrasts markedly with interaction though a narrow, local question answerer gate keeper.

Lewis Johnson's I-Doc is an example of delivering explanations of programs by hypertext browsing, using the World-wide Web.[6] Instead of producing large documents with general information about a system, I-Doc generates specific descriptions, sensitive to the user's work context and level of familiarity with the system. Explanation technology is used to guide the generation process.

I-Doc handles full natural language documentation, with arbitrary hypertext links. Navigating FR's, in contrast to documentation text, only requires a small set of link types, e.g. for FR primitives. On the other hand, FR should be more graphic, using diagrams instead of text, e.g. for CPD's, function definitions, and hierarchy visualization

4.4.3 Tool Design

We propose a tool that combines features of question answering, graphical hypertext navigation, and I-Doc.

The user sees the FR primarily as a hierarchy of CPD's, with controllable detail. This is like the presentation of the X/Open FR in the figures in Chapter 3. An initial view could give an overview of the FR, like Figure 3.1. Clicking on a box gives a diagram for the corresponding CPD, e.g. like the CPD figures in Chapter 3 such as Figure 3.2. Clicking on a function name gives its justification, as in Figure 1.4. Displaying such a justifications gives paths between CPD's, e.g. between PD6 and PD5 in that figure. Point and click interactions can be equivalent to asking questions (Figure 4.1).

Where it's meaningful, question answering semantics and interface is provided in parallel with point and click navigation. For example, a menu of relevant questions is displayed beside the graphical browser. When the cursor is on graphical elements, e.g. a function name, questions light up giving the meaning of possible mouse clicks, e.g. What and How questions based on the function name. Alternatively, navigation can be done by clicking on, or typing, a question instead of clicking in the diagram. Similarly, when the answer to a question is displayed graphically, the natural language answer can often be given in an answer box under the question. This immediately and constantly provides natural language explanation explicating the graphical explanation.

Such a tool could be implemented gradually, starting with the simplest capabilities and existing interfaces. For example, the CPD figures in Chapter 3 could be combined and browsed in a WWW demonstration.

4.5 Summary

This chapter discussed how explanations can be usefully obtained from FR-Rapide architecture FR's. Primitive explanation (and FR semantics) were described using the question answering paradigm. We gave a catalog of question types which may be answered and procedures for answering them. There were question types for all FR's and special question types for FR-Rapide. Asking and answering questions is similar to certain navigations in the FR. We contrasted the question answering and graphical navigation paradigms of explanation delivery. A practical explanation tool was described for for navigating the FR and providing needed answers and explanations on demand. It provides connected question answering and graphical hypertext navigation interfaces which complement each other.

Chapter 5

Evaluation and Discussion

This chapter evaluates and discusses the limitations, generality, benefits, and prospects for applying FR to architecture-based software engineering. We first introduce issues and a basis for evaluation. Then we discuss FR-Rapide with respect these criteria at three levels of generality: 1) the Rapide X/Open architecture, 2) other Rapide architectures and processes, and 3) general application to architecture-based methods. We conclude by reviewing our contributions and suggesting further work.

5.1 Evaluation Basis

We will take a top-down, benefit-based approach to evaluation. This means empirically considering the potential benefits of FR applications to architecture-based methods, and balancing them with costs and limitations. The resulting cost-benefit can be compared with alternative approaches for particular FR's, applications, and tasks.

5.1.1 FR Benefits

The general benefits of creating and using FR's for architectures were discussed in Introduction Section 1.4.2. Writing an FR creates and systematizes understanding. The extant FR is a notation which records and recalls this understanding for the author. Similarly, the FR is a notation for communication between the author and others. It serves the role of documentation, but with added formalized structure and conventions. Finally, the FR allows captured understanding to be manipulated, delivered, and exploited by automated tools and environments.

In all these roles, the captured understanding is beneficial if it makes it easier to perform architecture evolution tasks. This is described more precisely and operationally by the question answering paradigm. Question answering shows exactly what information can be provided to a human or automatic user from

an FR. This information is beneficial if it is necessary or useful for performing specific architecture evolution tasks. Chapter 4 gave types of questions that can be easily answered by an FR. For example, questions of the type "What does function do?" are answered by giving the state change caused by the function. The state change is described by precondition and postcondition states in a state language.

Therefore, beneficial answers and information can be delivered if the FR has useful content. We evaluate potential benefit by considering the content which can be represented in FR's, e.g. specific functions and state descriptions. FR is highly general for representing architecture understanding. We believe FR, in principle, can capture any useful architecture and software engineering causal content. There are no known theoretical counter-arguments. This working hypothesis of *FR representational completeness and generality* can be stated:

> FR can represent any human causal understanding used in software engineering.

Practical benefit depends on empirical issues, especially scalability, language, and validity. *Scalability* means that useful sized FR's can be created without practical problems of storage or access. *Language* issues involve the existence and semantics of adequate languages for FR state and function descriptions and justifications. Languages may range from informal to formally verifiable, depending on needs. *Validity* means that the FR is sufficiently correct for its intended purpose, within the constraints of its languages. These issues interact. For example, for given content, an FR with more detailed, formally verifiable state language requires more storage and access effort than natural language state descriptions, and is harder to create correctly.

Some beneficial FR's can obviously be created. Consider an architecture evolution task which depends on one page of documentation, e.g. documentation describing how architecture components change states in the requirements domain. This documentation can be represented in FR. The FR constitutes semi-formalized structured documentation. The FR increases the size of the documentation, perhaps to five pages, but it will still have practical scale. The FR state language will be natural language, and can have the same semantics and validity as the original. The FR can answer the same questions, and provide the same benefits to the task as the original documentation.

Example FR's are existence proofs for benefits. Practical benefits have been shown for example FR's in non-software domains. An FR for X/Open Rapide architecture is described in this report. This FR clearly has scale, language, and validity that is practically useful for tasks involving X/Open Poll-Decide, as discussed above and below. Practical benefits can be empirically investigated by creating more architecture FR's of varying size, language, and purpose.

5.1.2 FR Costs

Capturing understanding in FR can clearly be beneficial. However, the benefits must be worth the cost of creating and using the FR.

The obvious, seemingly unavoidable, cost is FR creation. As described in Chapter 1 and summarized in Table 1.1, creating an FR is a creative, laborious, expensive, and error-prone process. This cost can be seen as consisting of two components: 1) the cost of creating needed functional causal understanding, and 2) the cost of formatting this understanding in FR syntax. In many cases, 1) the cost of creating needed understanding can be fully or partially discounted. For example, the needed understanding may already exist, and it may also be in a form close to FR, e.g. related formal specifications. Or, if the understanding must be created, it is likely to be necessary or useful regardless of whether it is in FR or some other notation. In this case, the cost of creation can be reduced by the value of other uses of the created understanding.

The creation costs unique to FR include the costs of creating functional understanding of a form that would not be used otherwise, writing it in the FR language, and validating the FR. These costs vary widely, and depend on many situational and empirical factors. Note that creation is only performed once for a given FR, and therefore is a one-time overhead cost.

The cost of accessing the FR is also situational and empirical. Access may produce a net benefit. Free access is provided when the FR is read manually, like documentation. Other forms of interactive and automatic access, such as question answering and browsing, as discussed in Chapter 4, cost more but may offer additional benefits. Particularly, because FR formalizes key understanding primitives, it enables understanding to be accessed, manipulated, and exploited by a wide range of tools and environments. Automatic access can therefore be a net benefit instead of a cost. More broadly, appropriate access enabled by FR can create valuable synergies. For example, we said that the cost of creating understanding should not be charged against FR if the understanding would have been created otherwise, and/or has other uses. Having automatic access provided by FR may enhance those other uses, and provide additional applications beyond those which initially motivated FR creation.

5.1.3 Relative Cost-Benefit and Empirical Evaluation

Cost-benefit therefore depends on the value of benefits over time, versus the overhead of creating and using the FR. It depends on particular FR's, architectures, goals, tasks, applications, tools, environments, processes, people etc. Cost-benefit can only be determined empirically, in increasingly realistic applications and environments.

For both thought experiments, examples, and empirical evaluation, cost-benefit should be evaluated *relative to alternative approaches*. Presumably needed architecture evolution tasks will be performed in a particular environment, with particular people, process etc., with or without an architecture FR. FR cost-benefits should be compared with alternative existing or proposed approaches. The task fixes the scale, language, and validity needed, independent of FR. For example, above we said that an FR roughly equivalent to given short documentation can answer the same questions and provide at least the same benefits. A relative cost-benefit comparison would consider the cost of

reformatting the documentation in FR, and the cost and benefit of particular forms of access for the original documentation and the FR representation. In certain tasks, the FR gives easier, more useful access than the the documentation, e.g. hypertext browsing by function or isolating functional hierarchies under particular views.

So far we have one exploratory example of an architecture FR. Examples explore issues, show feasibility of construction and application, and give an upper bound data point for costs. The X/Open Poll-Decide example FR gives particular views, abstractions and languages. We evaluate this FR below. Then we generalize from our experience to other FR's for X/Open, FR's for other Rapide architectures and applications, and FR's for architecture-based software engineering with non-Rapide architectures and ADL's.

5.2 FR's for Rapide X/Open Architecture

5.2.1 Example X/Open Poll-Decide FR

The X/Open Poll-Decide FR and its authoring decisions are described in Chapter 3. Understanding X/Open and its design intentions is discussed at the end of Chapter 2. Writing this FR was an exploratory exercise, and this influenced the design designs. Because it was an exploratory exercise, many aspects were probably not typical, so we won't over-interpret the exercise. Here are some experiences and observations:

1. Authoring effort was high, in part because of the need to understand the architecture and its domains.

2. It was possible to understand and model the architecture with a finite state machine in the bottom level of the FR. This led to complete, simple descriptions with total states at other levels of the FR.

3. Abstract devices and components were not needed to control complexity.

4. The abstraction hierarchy reflected abstraction towards the chosen top-level of algorithmic intentions. Particular abstractions, e.g. distribution and concurrency removal, were somewhat independent of each other and abstraction order.

5. Intentions were complex, but they were organized into simple, local intentions, domains and languages by the chosen functional abstractions.

6. Given an understanding or view, it was relatively easy to form the FR. The understanding suggested the functions and state descriptions.

7. Other views and hierarchies could have been constructed to coexist in the same FR, as discussed below.

5.2.2 Other X/Open FR's

The X/Open FR was constructed with a general bias towards algorithmic understanding, e.g. for verifying and/or modifying the Two-Phase Commit implementation. Other tasks require views and abstractions not present or emphasized in the example FR. From our experience it seems possible to create FR's for other useful views and functions. Such FR's could exist independently. They could also be combined, in an FR which shared lower levels, and then branched upward at various levels to different "top-level" descriptions.

Similar Views and FR's

Here are some alternative views and FR's which seem to be constructible in a manner similar to the X/Open Poll-Decide FR:

1. Procedure Call Program – The FR displays procedure calling, and abstracts away concurrency and computed function. This could be used to visualize or modularize the calling structure.

2. Correctness Proof – The FR is a formal correctness proof of Poll-Decide's correct functioning in a larger atomicity proof. The state language and function justifications are formal specifications and proof steps. The FR hierarchy is the proof tree, and displays independent components to the extent that the implementation and proof are modular.

3. Poset Trace Interpretation – Chapter 4 described how questions can be answered about how the FR's intentions correspond to the poset behavior trace for a particular execution. Such correspondences could be permanently build into an FR, for one or more simulated executions. The correspondences could be given by a new annotation type or by functions with justifications specific for each execution.

4. X/Open Patient Billing System – In [11] Luckham and Kenney give another Rapide architecture, Patient Billing System, which incorporates the X/Open architecture. FR's can be constructed for this combined architecture, and other architectures embedding X/Open. They show the interface and functionality of X/Open in the larger architectures.

Problematic Views

On the other hand, we identified some views and understandings that could not be readily captured by FR. Some of these involved limits of static understanding, rather than FR alone.

Our FR and the others above are nominally based on understanding based on the single transaction assumption. Views involving multiple transactions were not attempted, in part because they are not necessary for algorithmic understanding of Poll-Decide. Presumably FR's could be created for reasoning about multiple transactions. They could involve interpretation of poset behaviors with

multiple transactions. They could also introduce states justified by the kinds of multiple transaction variants used in the transaction processing literature.

As one example, the X/Open architecture is parameterized so it describes a family of architectures. The family consists of architectures with an arbitrary number of Resource Managers. The parameter NumRMs specifies the number of Resource Managers at architecture generation time. Our X/Open FR was for an architecture with two Resource Managers. For a substantial number of RM's, the FR would explode in complexity. No way is now known to simply parameterize the FR for NumRMs, like the Rapide code is.

Similarly, Rapide is claimed to support dynamic architectures in the sense that architecture components can be created and destroyed at simulation time. It is challenging to represent understanding of such dynamism in FR (and other representations).

5.3 Rapide Architectures and Applications

5.3.1 Other Rapide Architectures

Here are some issues not encountered in the X/Open example FR which could affect the construction of FR's for other Rapide architectures.

1. Constructibility with finite number of states
2. Complete states vs. partial states and delocalization, complexity...
3. Use of Rapide constraints.

5.3.2 Rapide Applications

As discussed in Section 1.4.2, and above and below in this chapter, FR's are applicable to a wide variety of architecture tasks and tools. Tasks require understanding. When an FR captures and delivers that understanding it can be useful in an application. Furthermore, FR's can capture a wide range of human causal understanding.

The general architecture applications below mostly apply to Rapide tasks and tools. Here we will mention several specific Rapide applications. These are from the Rapide literature.[11] They would be good examples for studying the relative cost-benefit of FR-Rapide and possible tools.

The X/Open Patient Billing System shows how Rapide posets reveal a bug – failure of the Transaction Manager to poll all resource managers. FR's can be constructed to 1) reveal this bug statically, and 2) explain the bug revealed in the poset, using connections between the poset and intentions in an FR, including the failed coordination intention.

An FR can be used to plan simulations and record the rationale for particular simulation runs and analyses. For example, we can record understanding of why particular input may reveal a bug or correct function, in terms of abstract intentions.

The X/Open Patient Billing System was both a debugging example and an example of comparing two architectures, e.g. a local instance architecture against a reference architecture. FR's can be constructed to 1) compare architectures statically, based on differences in their FR's, and 2) explain comparison using Kenney's method of comparing mapped posets.

5.4 Architecture-Based Software Engineering

What are the generality and limitations of FR for architecture-based software engineering using ADL's and environments other than Rapide?

As we have repeatedly said, FR is broadly applicable, because many architecture tasks require causal understanding and explanation. FR applicability for an application to a given task or tool can be assessed by asking potential users:

"What explanation does your application need?"

If the needed explanation involves causal reasoning, FR could support it, in principle. We say this because we believe that FR can capture any causal understanding used in software engineering.

Specific applications could be need driven. They can involve various architecture processes, tasks, and users. These can each involve process types like design, maintenance, evolution, testing, analysis, verification, validation, and system implementation. For each of these, there are many specific tools and uses. Commonly mentioned categories include debugging, documentation (e.g. dynamic, structured, natural language, automatic...), explanation, simulation, rationale capture, question answering, trace interpretation and explanation, design verification, various kinds of inferences involving causal chains, and task-specific tools, e.g. an intention-based configurer for a domain-specific family of architectures and system products.

Perhaps it's easier to discuss what FR does not apply to. It certainly cannot create understanding that is otherwise impossible. FR, and any other representation of understanding, can only represent what can be known from the information available. One example of this above was the impossibility of representing behavior that depends on unavailable dynamic information.

More practically, FR is limited to causal processes. Temporal processes are usually causal processes. There are causal processes that are not acutely temporal. Chandrasekaran is refining these limitations of FR in forthcoming work on the foundations of FR.

Roughly, we say that architectures and ADL's are subject to causal understanding and FR representation if they are executable architectures. Architectures that consist only of static, structural relationships, e.g. module-connection descriptions, have no causal content, and FR is not relevant to them.

On the other hand, architectures that have temporal behavior are causal. This temporal behavior must be specified by some form of program. Therefore architecture understanding overlaps program understanding. Architectures

may have programming aspects less common in routine imperative, sequential programming, e.g. rules, distribution, and concurrency. But these aspects are also present in non-architectural programs. Conversely, imperative, sequential programming may be used to give architecture behavior, e.g. in implemented Rapide modules.

The point is, there is no clear division between work in program understanding and work in architecture understanding. We are exploring the use of FR to capture and exploit understanding in both program understanding and architecture understanding. The use of FR in program understanding is described in [4] and elsewhere. Insights from FR-Rapide architecture understanding have helped program understanding work, and vice versa.

5.5 Contributions and Future Work

The contributions of this work include:

1. A framework for discussing architecture understanding, its representation, and its applications.
2. FR-Rapide, a method of capturing and exploiting understanding of Rapide architectures.
3. An example FR representing hierarchical understanding of the X/Open architecture.
4. A model and semantics for FR explanation delivery based on the question answering paradigm, including important question types and answering procedures.
5. Design of a tool which delivers explanations from FR's, which combines question answering and graphic hypertext navigation.
6. Evaluation of with the example FR, and evaluation of the limitations, generality, benefits and prospects for applying FR to architecture-based software engineering.

Future work may include:

1. Development Examples – Constructing more architecture FR's to explore scalability, language and validity issues, and specific technical problems such as partial states and views, function justification syntax, and the use of abstract sub-devices.
2. Demonstrate Application Benefit – Start relative cost-benefit empirical studies, perhaps using existing Rapide examples and tools, as described above.
3. Tool Development – Start developing and testing an explanation tool, perhaps using some of of the design given above.

Acknowledgments

The Stanford Rapide project, including David Luckham, Larry Augustin and John Kenney, provided helpful insights and information.

Bibliography

[1] Dean Allemang and B. Chandrasekaran. Functional representation and program debugging. In *PROCEEDINGS of the 6TH ANNUAL KNOWLEDGE-BASED SOFTWARE ENGINEERING CONFERENCE.* IEEE Computer Society Press, 1991.

[2] B. Chandrasekaran. Functional representation and causal processes. In Marshall Yovits, editor, *ADVANCES in COMPUTERS.* Academic Press, 1994.

[3] David Garlan and Dewayne E. Perry. Introduction to the special issue on software architecture. *IEEE Transactions on Software Engineering*, 21(4), April 1995.

[4] John Hartman and B. Chandrasekaran. Functional representation and understanding of software: Technology and application. In *5TH ANNUAL DUAL-USE TECHNOLOGIES and APPLICATIONS CONFERENCE.* Mohawk IEEE and Rome Lab, May 1995. Utica, New York.

[5] Y. Iwasaki and B. Chandrasekaran. Design verification through function- and behavior-oriented representations. In *Proceedings of the Conference on Artificial Intelligence and Design*, 1992.

[6] W. Lewis Johnson and Ali Erdem. Interactive explanation of software systems. In *Proceedings KBSE'95 – The Tenth Knowledge-Based Software Engineering Conference.* IEEE Computer Society Press, November 12-15 1995. Boston, Mass.

[7] John R. Josephson. Technical note on formalizing functional representation. In J. Hodges, editor, *AAAI-94 WORKSHOP on REPRESENTING and REASONING ABOUT DEVICE FUNCTION.* AAAI, 1994. Seattle, Washington.

[8] John J. Kenney. *EXECUTABLE FORMAL MODELS of DISTRIBUTED TRANSACTION SYSTEMS BASED ON EVENT PROCESSING.* PhD thesis, Stanford University, June 1995.

[9] Paul Kogut and Paul Clements. The software architecture renaissance. *CrossTalk*, 7(11):20–23, November 1994.

[10] Susan T. Korda. *USING FUNCTIONAL REPRESENTATION FOR SMART SIMULATION OF DEVICES.* PhD thesis, The Ohio State University, Dept. of Computer and Information Science, 1993.

[11] David C. Luckham, John J. Kenney, Larry M. Augustin, James Vera, Doug Bryan, and Walter Mann. Specification and analysis of system architecture using Rapide. *IEEE Transactions on Software Engineering*, 21(4):336–355, April 1995.

[12] David C. Luckham and James Vera. An event-based architecture definition language. *IEEE Transactions on Software Engineering*, 21(9):717–734, September 1995.

☆U.S. GOVERNMENT PRINTING OFFICE: 1998-610-130-61165

DISTRIBUTION LIST

addresses	number of copies
DOUGLAS WHITE RL/C3CB 525 BROOKS ROAD ROME NY 13441-4505	20
DR. B. CHANDRASEKASAN DEPT. OF COMPUTER & INFORM SCIENCE THE OHIO STATE UNIVERSITY 2015 NEIL AVENUE COLUMBUS, OH 43210-1277	5
ROME LABORATORY/SUL TECHNICAL LIBRARY 26 ELECTRONIC PKY ROME NY 13441-4514	1
ATTENTION: DTIC-OCC DEFENSE TECHNICAL INFO CENTER 8725 JOHN J. KINGMAN ROAD, STE 0944 FT. BELVOIR, VA 22060-6218	2
ADVANCED RESEARCH PROJECTS AGENCY 3701 NORTH FAIRFAX DRIVE ARLINGTON VA 22203-1714	1
RELIABILITY ANALYSIS CENTER 201 MILL ST. ROME NY 13440-8200	1
ROME LABORATORY/C3AB 525 BROOKS RD ROME NY 13441-4505	1
ATTN: GWEN NGUYEN GIDEP P.O. BOX 8000 CORONA CA 91718-8000	1

AFIT ACADEMIC LIBRARY/LDEE
2950 P STREET
AREA B, BLDG 642
WRIGHT-PATTERSON AFB OH 45433-7765 — 1

ATTN: R.L. DENISON
WRIGHT LABORATORY/MLPO, BLDG. 651
3005 P STREET, STE 6
WRIGHT-PATTERSON AFB OH 45433-7707 — 1

ATTN: GILBERT G. KUPERMAN
AL/CFHI, BLDG. 248
2255 H STREET
WRIGHT-PATTERSON AFB OH 45433-7022 — 1

ATTN: TECHNICAL DOCUMENTS CENTER
OL AL HSC/HRG
2698 G STREET
WRIGHT-PATTERSON AFB OH 45433-7604 — 1

AIR UNIVERSITY LIBRARY (AUL/LSAD)
600 CHENNAULT CIRCLE
MAXWELL AFB AL 36112-6424 — 1

US ARMY SSDC
P.O. BOX 1500
ATTN: CSSD-IM-PA
HUNTSVILLE AL 35807-3801 — 1

COMMANDING OFFICER
NCCOSC RDT&E DIVISION
ATTN: TECHNICAL LIBRARY,CODE D0274
53560 HULL STREET
SAN DIEGO CA 92152-5001 — 1

NAVAL AIR WARFARE CENTER
WEAPONS DIVISION
CODE 4BL000D
1 ADMINISTRATION CIRCLE
CHINA LAKE CA 93555-6100 — 1

SPACE & NAVAL WARFARE SYSTEMS CMD
ATTN: PMW163-1 (R. SKIANO)RM 1044A
53560 HULL ST.
SAN DIEGO, CA 92152-5002 — 2

SPACE & NAVAL WARFARE SYSTEMS COMMAND, EXECUTIVE DIRECTOR (PD13A)
ATTN: MR. CARL ANDRIANI
2451 CRYSTAL DRIVE
ARLINGTON VA 22245-5200 — 1

COMMANDER, SPACE & NAVAL WARFARE SYSTEMS COMMAND (CODE 32)
2451 CRYSTAL DRIVE
ARLINGTON VA 22245-5200 — 1

CDR, US ARMY MISSILE COMMAND
REDSTONE SCIENTIFIC INFORMATION CTR
ATTN: AMSMI-RD-CS-R, DOCS
REDSTONE ARSENAL AL 35898-5241 — 2

ADVISORY GROUP ON ELECTRON DEVICES
SUITE 500
1745 JEFFERSON DAVIS HIGHWAY
ARLINGTON VA 22202 — 1

REPORT COLLECTION, CIC-14
MS P364
LOS ALAMOS NATIONAL LABORATORY
LOS ALAMOS NM 87545 — 1

AEDC LIBRARY
TECHNICAL REPORTS FILE
100 KINDEL DRIVE, SUITE C211
ARNOLD AFB TN 37389-3211 — 1

COMMANDER
USAISC
ASHC-IMD-L, BLDG 61801
FT HUACHUCA AZ 85613-5000 — 1

US DEPT OF TRANSPORTATION LIBRARY
FB10A, M-457, RM 930
800 INDEPENDENCE AVE, SW
WASH DC 22591 — 1

AWS TECHNICAL LIBRARY
859 BUCHANAN STREET, RM. 427
SCOTT AFB IL 62225-5118 — 1

AFIWC/MSY
102 HALL BLVD, STE 315
SAN ANTONIO TX 78243-7016 — 1

SOFTWARE ENGINEERING INSTITUTE
CARNEGIE MELLON UNIVERSITY
4500 FIFTH AVENUE
PITTSBURGH PA 15213 — 1

NSA/CSS
K1
FT MEADE MD 20755-6000 — 1

ATTN: OM CHAUHAN
DCMC WICHITA
271 WEST THIRD STREET NORTH
SUITE 6000
WICHITA KS 67202-1212 — 1

PHILLIPS LABORATORY
PL/TL (LIBRARY)
5 WRIGHT STREET
HANSCOM AFB MA 01731-3004 — 1

ATTN: EILEEN LADUKE/D460
MITRE CORPORATION
202 BURLINGTON RD
BEDFORD MA 01730 — 1

OUSD(P)/DTSA/DUTD
ATTN: PATRICK G. SULLIVAN, JR.
400 ARMY NAVY DRIVE
SUITE 300
ARLINGTON VA 22202 — 2

DR. ROBERT PARKER
DARPA/ITO
3701 NORTH FAIRFAX DRIVE
ARLINGTON VA 22203-1714 — 1

DR. GARY KOOB
DARPA/ITO
3701 NORTH FAIRFAX DRIVE
ARLINGTON VA 22203-1714 — 1

DR. ROBERT LUCAS 1
DARPA/ITO
3701 NORTH FAIRFAX DRIVE
ARLINGTON VA 22203-1714

DR. DAVID GUNNING 1
DARPA/ITO
3701 NORTH FAIRFAX DRIVE
ARLINGTON VA 22203-1714

JOHN SALASIN 1
DARPA
3701 N. FAIRFAX DRIVE
ARLINGTON, VA 22203-1714

HOWARD SHROBE 1
DARPA
3701 N. FAIRFAX DRIVE
ARLINGTON, VA 22203-1714

HELEN GILL 1
DARPA
3701 N. FAIRFAX DRIVE
ARLINGTON, VA 22203-1714

ROBERT LADDAGA 1
DARPA
3701 N. FAIRFAX DRIVE
ARLINGTON, VA 22203-1714

MARK SHULTZ 1
TRI-STATE RESEARCH
P.O. BOX 2194
NATICK MA 01760

DR. TANYA KORELSKY 1
COGENTEX, INC.
840 HANSHAW ROAD, SUITE 5
ITHACA, NY 14850-1589

BARRY HANTMAN 1
RAYTHEON
MISSLE SYSTEMS DIVISION
50 APPLE HILL DRIVE
TEWKSBURY, MA 01876-0901

DR. JOSEPH HINTZ 1
RAYTHEON
MISSLE SYSTEMS DIVISION
50 APPLE HILL DRIVE
TEWKSBURY, MA 01876-0901

TOM GREENE 1
EXPERSOFT
5825 OBERLIN DRIVE
SAN DIEGO, CA 92121